

Signals and Communication Technology

Uwe Meyer-Baese

Digital Signal Processing with Field Programmable Gate Arrays

Fourth Edition

 Springer

Signals and Communication Technology

For further volumes:
<http://www.springer.com/series/4748>

Uwe Meyer-Baese

Digital Signal Processing with Field Programmable Gate Arrays

Fourth Edition



Springer

Uwe Meyer-Baese
Department of Electrical and Computer
Engineering
Florida State University
Tallahassee, FL
USA

ISSN 1860-4862 ISSN 1860-4870 (electronic)
ISBN 978-3-642-45308-3 ISBN 978-3-642-45309-0 (eBook)
DOI 10.1007/978-3-642-45309-0
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014939295

© Springer-Verlag Berlin Heidelberg 2014

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. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law. 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.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To

Anke, Lisa, and my Parents

Preface to First Edition

Field-programmable gate arrays (FPGAs) are on the verge of revolutionizing digital signal processing in the manner that programmable digital signal processors (PDSPs) did nearly two decades ago. Many front-end digital signal processing (DSP) algorithms, such as FFTs, FIR or IIR filters, to name just a few, previously built with ASICs or PDSPs, are now most often replaced by FPGAs. Modern FPGA families provide DSP arithmetic support with fast-carry chains (Xilinx Virtex, Altera FLEX) that are used to implement multiply-accumulates (MACs) at high speed, with low overhead and low costs [1]. Previous FPGA families have most often targeted TTL “glue logic” and did not have the high gate count needed for DSP functions. The efficient implementation of these front-end algorithms is the main goal of this book.

At the beginning of the twenty-first century we find that the two programmable logic device (PLD) market leaders (Altera and Xilinx) both report revenues greater than US\$1 billion. FPGAs have enjoyed steady growth of more than 20% in the last decade, outperforming ASICs and PDSPs by 10%. This comes from the fact that FPGAs have many features in common with ASICs, such as reduction in size, weight, and power dissipation, higher throughput, better security against unauthorized copies, reduced device and inventory cost, and reduced board test costs, and claim advantages over ASICs, such as a reduction in development time (rapid prototyping), in-circuit reprogrammability, lower NRE costs, resulting in more economical designs for solutions requiring less than 1000 units. Compared with PDSPs, FPGA design typically exploits parallelism, e.g., implementing multiple multiply-accumulate calls efficiency, e.g., zero product-terms are removed, and pipelining, i.e., each LE has a register, therefore pipelining requires no additional resources.

Another trend in the DSP hardware design world is the migration from graphical design entries to hardware description language (HDL). Although many DSP algorithms can be described with “signal flow graphs,” it has been found that “code reuse” is much higher with HDL-based entries than with graphical design entries. There is a high demand for HDL design engineers and we already find undergraduate classes about logic design with HDLs [2]. Unfortunately *two* HDL languages are popular today. The US west coast and

Asia area prefer Verilog, while US east coast and Europe more frequently use VHDL. For DSP with FPGAs both languages seem to be well suited, although some VHDL examples are a little easier to read because of the supported signed arithmetic and multiply/divide operations in the IEEE VHDL 1076-1987 and 1076-1993 standards. The gap is expected to disappear after approval of the Verilog IEEE standard 1364-1999, as it also includes signed arithmetic. Other constraints may include personal preferences, EDA library and tool availability, data types, readability, capability, and language extensions using PLIs, as well as commercial, business, and marketing issues, to name just a few [3]. Tool providers acknowledge today that both languages have to be supported and this book covers examples in both design languages.

We are now also in the fortunate situation that “baseline” HDL compilers are available from different sources at essentially no cost for educational use. We take advantage of this fact in this book. It includes a CD-ROM with Altera’s newest **MaxPlusII** software, which provides a complete set of design tools, from a content-sensitive editor, compiler, and simulator, to a bitstream generator. All examples presented are written in VHDL and Verilog and should be easily adapted to other propriety design-entry systems. Xilinx’s “Foundation Series,” ModelTech’s ModelSim compiler, and Synopsys FC2 or FPGA Compiler should work without any changes in the VHDL or Verilog code.

The book is structured as follows. The first chapter starts with a snapshot of today’s FPGA technology, and the devices and tools used to design state-of-the-art DSP systems. It also includes a detailed case study of a frequency synthesizer, including compilation steps, simulation, performance evaluation, power estimation, and floor planning. This case study is the basis for more than 30 other design examples in subsequent chapters. The second chapter focuses on the computer arithmetic aspects, which include possible number representations for DSP FPGA algorithms as well as implementation of basic building blocks, such as adders, multipliers, or sum-of-product computations. At the end of the chapter we discuss two very useful computer arithmetic concepts for FPGAs: distributed arithmetic (DA) and the CORDIC algorithm. Chapters 3 and 4 deal with theory and implementation of FIR and IIR filters. We will review how to determine filter coefficients and discuss possible implementations optimized for size or speed. Chapter 5 covers many concepts used in multirate digital signal processing systems, such as decimation, interpolation, and filter banks. At the end of Chap. 5 we discuss the various possibilities for implementing wavelet processors with two-channel filter banks. In Chap. 6, implementation of the most important DFT and FFT algorithms is discussed. These include Rader, chirp- z , and Goertzel DFT algorithms, as well as Cooley–Tuckey, Good–Thomas, and Winograd FFT algorithms. In Chap. 7 we discuss more specialized algorithms, which seem to have great potential for improved FPGA implementation when compared with PDSPs. These algorithms include number theoretic transforms, algorithms for cryp-

tography and errorcorrection, and communication system implementations. The appendix includes an overview of the VHDL and Verilog languages, the examples in Verilog HDL, and a short introduction to the utility programs included on the CD-ROM.

Acknowledgements. This book is based on an FPGA communications system design class I taught for four years at the Darmstadt University of Technology; my previous (German) books [4, 5]; and more than 60 Masters thesis projects I have supervised in the last 10 years at Darmstadt University of Technology and the University of Florida at Gainesville. I wish to thank all my colleagues who helped me with critical discussions in the lab and at conferences. Special thanks to: M. Achery, D. Achilles, F. Bock, C. Burrus, D. Chester, D. Childers, J. Conway, R. Crochiere, K. Damm, B. Delguette, A. Dempster, C. Dick, P. Duhamel, A. Drolshagen, W. Endres, H. Eveking, S. Foo, R. Games, A. Garcia, O. Ghitza, B. Harvey, W. Hilberg, W. Jenkins, A. Laine, R. Laur, J. Mangen, J. Massey, J. McClellan, F. Ohl, S. Orr, R. Perry, J. Ramirez, H. Scheich, H. Scheid, M. Schroeder, D. Schulz, F. Simons, M. Soderstrand, S. Stearns, P. Vaidyanathan, M. Vetterli, H. Walter, and J. Wietzke.

I would like to thank my students for the innumerable hours they have spent implementing my FPGA design ideas. Special thanks to: D. Abdolrahimi, E. Allmann, B. Annamaier, R. Bach, C. Brandt, M. Brauner, R. Bug, J. Burros, M. Burschel, H. Diehl, V. Dierkes, A. Dietrich, S. Dworak, W. Fieber, J. Guyot, T. Hattermann, T. Häuser, H. Hausmann, D. Herold, T. Heute, J. Hill, A. Hundt, R. Huthmann, T. Irmler, M. Katzenberger, S. Kenne, S. Kerkmann, V. Kleipa, M. Koch, T. Krüger, H. Leitel, J. Maier, A. Noll, T. Podzimek, W. Praefcke, R. Resch, M. Rösch, C. Scheerer, R. Schimpf, B. Schlanske, J. Schleichert, H. Schmitt, P. Schreiner, T. Schubert, D. Schulz, A. Schuppert, O. Six, O. Spiess, O. Tamm, W. Trautmann, S. Ullrich, R. Watzel, H. Wech, S. Wolf, T. Wolf, and F. Zahn.

For the English revision I wish to thank my wife Dr. Anke Meyer-Bäse, Dr. J. Harris, Dr. Fred Taylor from the University of Florida at Gainesville, and Paul DeGroot from Springer.

For financial support I would like to thank the DAAD, DFG, the European Space Agency, and the Max Kade Foundation.

If you find any errata or have any suggestions to improve this book, please contact me at Uwe.Meyer-Baese@ieee.org or through my publisher.

Tallahassee, May 2001

Uwe Meyer-Bäse

Preface to Second Edition

A new edition of a book is always a good opportunity to keep up with the latest developments in the field and to correct some errors in previous editions. To do so, I have done the following for this second edition:

- Set up a web page for the book at the following URL:
www.eng.fsu.edu/~umb
The site has additional information on DSP with FPGAs, useful links, and additional support for your designs, such as code generators and extra documentation.
- Corrected the mistakes from the first edition. The errata for the first edition can be downloaded from the book web page.
- A total of approximately 100 pages have been added to the new edition. The major new topics are:
 - The design of serial and array dividers
 - The description of a complete floating-point library
 - A new Chap. 8 on adaptive filter design
- Altera's current student version has been updated from 9.23 to 10.2 and all design examples, size and performance measurements, i.e., many tables and plots have been compiled for the EPF10K70RC240-4 device that is on Altera's university board UP2. Altera's UP1 board with the EPF10K20RC240-4 has been discontinued.
- A solution manual for the first edition (with more than 65 exercises and over 33 additional design examples) is available from Amazon. Some additional (over 25) new homework exercises are included in the second edition.

Acknowledgements. I would like to thank my colleagues and students for the feedback to the first edition. It helped me to improve the book. Special thanks to: P. Ashenden, P. Athanas, D. Belc, H. Butterweck, S. Connors, G. Couturier, P. Costa, J. Hamblen, M. Horne, D. Hyde, W. Li, S. Lowe, H. Natarajan, S. Rao, M. Rupp, T. Sexton, D. Sunkara, P. Tomaszewicz, F. Verahrami, and Y. Yunhua.

From Altera, I would like to thank B. Esposito, J. Hanson, R. Maroccia, T. Mossadak, and A. Acevedo (now with Xilinx) for software and hardware support and the permission to include datasheets and MaxPlus II on the CD of this book.

From my publisher (Springer-Verlag) I would like to thank P. Jantzen, F. Holzwarth, and Dr. Merkle for their continuous support and help over recent years.

I feel excited that the first edition was a big success and sold out quickly. I hope you will find this new edition even more useful. I would also be grateful, if you have any suggestions for how to improve the book, if you would e-mail me at Uwe.Meyer-Baese@ieee.org or contact me through my publisher.

Tallahassee, October 2003

Uwe Meyer-Bäse

Preface to Third Edition

Since FPGAs are still a rapidly evolving field, I am very pleased that my publisher Springer Verlag gave me the opportunity to include new developments in the FPGA field in this third edition. A total of over 150 pages of new ideas and current design methods have been added. You should find the following innovations in this third edition:

- 1) Many FPGAs now include embedded 18×18 -bit multipliers and it is therefore recommended to use these devices for DSP-centered applications since an embedded multiplier will save many LEs. The Cyclone II EP2C35F672C6 device for instance, used in all the examples in this edition, has 35 18×18 -bit multipliers.
- 2) MaxPlus II software is no longer updated and new devices such as the Stratix or Cyclone are only supported in Quartus II. All old and new examples in the book are now compiled with Quartus 6.0 for the Cyclone II EP2C35F672C6 device. Starting with Quartus II 6.0 integers are by default initialized with the smallest negative number (similar to with the ModelSim simulator) rather than zero and the verbatim 2/e examples will therefore not work with Quartus II 6.0. Tcl scripts are provided that allow the evaluation of all examples with other devices too. Since downloading Quartus II can take a long time the book CD includes the web version 6.0 used in the book.
- 3) The new device features now also allow designs that use many MAC calls. We have included a new section (2.9) on MAC-based function approximation for trigonometric, exponential, logarithmic, and square root.
- 4) To shorten the time to market further FPGA vendors offer intellectual property (IP) cores that can be easily included in the design project. We explain the use of IP blocks for NCOs, FIR filters, and FFTs.
- 5) Arbitrary sampling rate change is a frequent problem in multirate systems and we describe in Sect. 5.6 several options including B-spline, MOMS, and Farrow-type converter designs.
- 6) FPGA-based microprocessors have become an important IP block for FPGA vendors. Although they do not have the high performance of a custom algorithm design, the software implementation of an algorithm with a μ P usually needs much less resources. A complete new chapter (9) covers many aspects from software tool to hard- and softcore μ Ps. A

complete example processor with an assembler and C compiler is developed.

- 7)** A total of 107 additional problems have been added and a solution manual will be available later from www.amazon.com at a not-for-profit price.
- 8)** Finally a special thank you goes to Harvey Hamel who discovered many errors that have been summarized in the errata for 2/e that is posted at the book homepage

Acknowledgements. Again many colleagues and students have helped me with related discussions and feedback to the second edition, which helped me to improve the book. Special thanks to:

P. Athanas, M. Bolic, C. Bentancourt, A. Canosa, S. Canosa, C. Chang, J. Chen, T. Chen, J. Choi, A. Comba, S. Connors, J. Coutu, A. Dempster, A. El-wakil, T. Felderhoff, O. Gustafsson, J. Hallman, H. Hamel, S. Hashim, A. Hoover, M. Karlsson, K. Khanachandani, E. Kim, S. Kulkarni, K. Lenk, E. Manolakos, F. Mirzapour, S. Mitra, W. Moreno, D. Murphy, T. Meißner, K. Nayak, H. Ningxin, F. von Münchow-Pohl, H. Quach, S. Rao, S. Stepanov, C. Suslowicz, M. Unser J. Vega-Pineda, T. Zeh, E. Zurek

I am particular thankful to P. Thévenaz from EPFL for help with the newest developments in arbitrary sampling rate changers.

My colleagues from the ISS at RHTH Aachen I would like to thank for their time and efforts to teach me LISA during my Humboldt award sponsored summer research stay in Germany. Special thanks go to H. Meyr, G. Ascheid, R. Leupers, D. Kammler, and M. Witte.

From Altera, I would like to thank B. Esposito, R. Maroccia, and M. Phipps for software and hardware support and permission to include datasheets and Quartus II software on the CD of this book. From Xilinx I like to thank for software and hardware support of my NSF CCLI project J. Weintraub, A. Acevedo, A. Vera, M. Pattichis, C. Sepulveda, and C. Dick.

From my publisher (Springer-Verlag) I would like to thank Dr. Baumann, Dr. Merkle, M. Hanich, and C. Wolf for the opportunity to produce an even more useful third edition.

I would be very grateful if you have any suggestions for how to improve the book and would appreciate an e-mail to Uwe.Meyer-Baese@ieee.org or through my publisher.

Tallahassee, May 2007

Uwe Meyer-Bäse

Preface to Fourth Edition

In recent years FPGAs have increased in complexity so that we can now build large DSP systems with a single FPGA. Modern devices now have hundreds of embedded multipliers and large on-chip memories. Since the previous books were written mainly to optimize the size of the design, system design questions now become more important. A good study of such issues can be carried out with larger tasks such as PCA or ICA algorithms, image and video processing systems, or a new 256-point FFT design discussed in this edition. A total of over 150 pages – including 11 new system designs ideas – have been added to this new edition, some requiring more than 100 embedded multipliers. You should find the following innovations in this fourth edition:

- 1) Simulation in HDL in the book is now done with the powerful MODELSIM simulator for Altera devices and the ISIM simulator for the Xilinx designs.
- 2) For the system designs many test bench data are now provided using MATLAB or SIMULINK.
- 3) System level designs using the VHDL-2008 new fixed-point and floating-point IEEE libraries are introduced.
- 4) All-pass IIR filter designs in direct form, BiQuads, lattices, and wave digital filters are compared.
- 5) Independent component analysis (ICA) and principle component analysis (PCA) algorithms are implemented.
- 6) Speech and audio compression methods from A-law, ADPCM to MP3 are discussed and implemented in HDL.
- 7) Image processing algorithms for edge detection and median filtering using HDL and embedded microprocessors are discussed.
- 8) Video processing for motion compensation using microprocessors with custom instructions are discussed.
- 9) Plans exist to provide design examples for the SIMULINK toolbox from Altera and Xilinx as well as support for Xilinx ISE and ISIM simulation later as additional CDs available through www.amazon.com.
- 10) Updates and bug fix report will be posted at the author's webpage, that is, at the time of writing, at www.eng.fsu.edu/~umb

Acknowledgements. Again many colleagues helped me with related discussions and feedback to the third edition, which helped me to improve the book. Special thanks to:

R. Adhami, M. Abd-El-Hameed, C. Allen, S. Amalkar, A. Andrawis, G. Ascheid, P. Athanas, S. Badave, R. Badeau, S. Bald, A. Bardakcioglu, P. Bendixen, C. Bentancourth, R. Bhakthavatchalu, G. Birkelbach, T. Borsodi, F. Casado, E. Castillo, O. Calvo, P. Cayuela, A. Celebi, C.-H. Chang, A. Chanerley, K. Chapman, I. Chiorescu, G. Connally, S. Connors, J. Coutu, S. Cox, S. David, R. Deka, A. Dempster, J. Domingo, A. Elias, F. Engel, R. van Engelen, H. Fan, S. Foo, T. Fox, M. Frank, J. Gallagher, A. Garcia, M. Gerhardt, A. Ghalame, G. Glandon, S. Grunwald, A. Guerrero, W. Guolin, O. Gustafsson, H. Hamel, S. Hashim, S. Hedayat, D. Hodali, S. Hong, K. Huang, F. Iqbal, R. Jadhav, D. Jiang, D. Kammler, K. Karuri, T. Knudsen, M. Koepnick, F. Koushanfar, M. Kumm, M. Krishna, H. LeFevre, R. Leupers, S. Liljeqvist, A. Littek, A. Lloris, M. Luqman, V. Madan, M. Manikandan, J. Mark, B. McKenzie, H. Meyr, P. Mishra, A. Mitra, I. Miu, J. Moorhead, S. Moradi, F. Munsche, Z. Navabi, L. Oniciuc, B. Parhami, S. Park, L. Parrilla, V. Pedroni, R. Pereira, R. Perry, A. Pierce, F. Poderico, G. Prinz, D. Raic, N. Rafla, S. Rao, N. Relia, F. Rice, D. Romero, D. Sarma, P. Sephra, W. Sheng, T. Taguchi, N. Trong, C. Unterrieder, G. Wall, G. Wang, Y. Wang, R. Wei, S. Weihua, J. Wu, J. Xu, O. Zavala-Romero, P. Zipf, D. Zhang, L. Zhang, M. Zhang,

Special thanks also to the students from my Spring EEL5722 DSP with FPGAs class. Thanks to Nick Stroupe for the DWT de-noising project work, Ye Yang for the LPC project work, Soumak Mookherjee for the 256-point FFT project, Naren Nagaraj for the all-pass filter project work, Venkata Pothavajhala for the BiQuad floating-point design, Haojun Yang for the lattice filter design project, and Crispin Odom for her ICA project and MS thesis work.

I am particular thankful to Guillermo Botella and Diego González from the University of Madrid for help with the image and video processing chapter.

Special thank also to David Bishop and Huibert Lincklaen Arriens for permission to include their libraries on the book CD.

From Altera, I would like to thank Ben Esposito, M. Phipps, Ralene Maroccia, Blair Fort, and Stephen Brown for software and hardware support. From Xilinx support I would like to thank A. Vera, M. Pattichis, Craig Kief, and Parimal Patel.

From my publisher (Springer-Verlag) I would like to thank Dr. Baumann, for his patience and the opportunity to update the book.

I would be very grateful if you have any suggestions on how to improve the book and would appreciate e-mail to Uwe.Meyer-Baese@ieee.org or through my publisher.

Tallahassee, January 2014

Uwe Meyer-Bäse

Contents

Preface to First Edition	VII
Preface to Second Edition	XI
Preface to Third Edition.....	XIII
Preface to Fourth Edition	XV
1. Introduction	1
1.1 Overview of Digital Signal Processing (DSP)	1
1.2 FPGA Technology	3
1.2.1 Classification by Granularity	3
1.2.2 Classification by Technology	6
1.2.3 Benchmark for FPLs	7
1.3 DSP Technology Requirements	12
1.3.1 FPGA and Programmable Signal Processors	14
1.4 Design Implementation	15
1.4.1 FPGA Structure	20
1.4.2 The Altera EP4CE115F29C7	23
1.4.3 Case Study: Frequency Synthesizer	32
1.4.4 Design with Intellectual Property Cores	40
Exercises	47
2. Computer Arithmetic.....	57
2.1 Introduction	57
2.2 Number Representation	58
2.2.1 Fixed-Point Numbers	58
2.2.2 Unconventional Fixed-Point Numbers	61
2.2.3 Floating-Point Numbers	75
2.3 Binary Adders	79
2.3.1 Pipelined Adders	81
2.3.2 Modulo Adders	85
2.4 Binary Multipliers	86
2.4.1 Multiplier Blocks	89

2.5	Binary Dividers	93
2.5.1	Linear Convergence Division Algorithms	95
2.5.2	Fast Divider Design.....	100
2.5.3	Array Divider	105
2.6	Fixed-Point Arithmetic Implementation	106
2.7	Floating-Point Arithmetic Implementation	109
2.7.1	Fixed-Point to Floating-Point Format Conversion.....	110
2.7.2	Floating-Point to Fixed-Point Format Conversion.....	111
2.7.3	Floating-Point Multiplication	112
2.7.4	Floating-Point Addition	113
2.7.5	Floating-Point Division	115
2.7.6	Floating-Point Reciprocal	116
2.7.7	Floating-Point Operation Synthesis	118
2.7.8	Floating-Point Synthesis Results	122
2.8	Multiply-Accumulator (MAC) and Sum of Product (SOP) ..	124
2.8.1	Distributed Arithmetic Fundamentals	125
2.8.2	Signed DA Systems	129
2.8.3	Modified DA Solutions	130
2.9	Computation of Special Functions Using CORDIC.....	131
2.9.1	CORDIC Architectures	135
2.10	Computation of Special Functions using MAC Calls.....	141
2.10.1	Chebyshev Approximations	142
2.10.2	Trigonometric Function Approximation	143
2.10.3	Exponential and Logarithmic Function Approximation	152
2.10.4	Square Root Function Approximation	159
2.10	Fast Magnitude Approximation	165
	Exercises	168
3.	Finite Impulse Response (FIR) Digital Filters	179
3.1	Digital Filters	179
3.2	FIR Theory.....	180
3.2.1	FIR Filter with Transposed Structure	181
3.2.2	Symmetry in FIR Filters	184
3.2.3	Linear-phase FIR Filters	185
3.3	Designing FIR Filters	187
3.3.1	Direct Window Design Method	187
3.3.2	Equiripple Design Method	190
3.4	Constant Coefficient FIR Design	192
3.4.1	Direct FIR Design	192
3.4.2	FIR Filter with Transposed Structure	196
3.4.3	FIR Filters Using Distributed Arithmetic.....	204
3.4.4	IP Core FIR Filter Design	215
3.4.5	Comparison of DA- and RAG-Based FIR Filters	217
	Exercises	219

4.	Infinite Impulse Response (IIR) Digital Filters	225
4.1	IIR Theory	228
4.2	IIR Coefficient Computation	231
4.2.1	Summary of Important IIR Design Attributes	233
4.3	IIR Filter Implementation	234
4.3.1	Finite Wordlength Effects	238
4.3.2	Optimization of the Filter Gain Factor	239
4.4	Fast IIR Filter	240
4.4.1	Time-domain Interleaving	241
4.4.2	Clustered and Scattered Look-Ahead Pipelining	243
4.4.3	IIR Decimator Design	246
4.4.4	Parallel Processing	246
4.4.5	IIR Design Using RNS	250
4.5	Narrow Band IIR Filter	250
4.5.1	Narrow Band Design Example	251
4.5.2	Cascade Second Order Systems Narrow Band Filter Design	259
4.5.3	Parallel Second Order Systems Narrow Band Filter Design	263
4.5.4	Lattice Design of Narrow Band IIR Filter	271
4.5.5	Wave Digital Filter Design of Narrow Band IIR Filter	280
4.6	All-Pass Filter Design of Narrow Band IIR Filter	287
4.6.1	All-Pass Wave Digital Filter Design of Narrow Band IIR Filter	289
4.6.2	All-Pass Lattice Design of Narrow Band IIR Filter	293
4.6.3	All-Pass Direct Form Design of Narrow Band Filter	294
4.6.4	All-Pass Cascade BiQuad of Narrow Band Filter	295
4.6.5	All-Pass Parallel BiQuad of Narrow Band Filter	295
	Exercises	299
5.	Multirate Signal Processing	305
5.1	Decimation and Interpolation	305
5.1.1	Noble Identities	306
5.1.2	Sampling Rate Conversion by Rational Factor	308
5.2	Polyphase Decomposition	309
5.2.1	Recursive IIR Decimator	314
5.2.2	Fast-running FIR Filter	315
5.3	Hogenauer CIC Filters	317
5.3.1	Single-Stage CIC Case Study	317
5.3.2	Multistage CIC Filter Theory	320
5.3.3	Amplitude and Aliasing Distortion	325
5.3.4	Hogenauer Pruning Theory	328
5.3.5	CIC RNS Design	333
5.3.6	CIC Compensation Filter Design	334
5.4	Multistage Decimator	337

5.4.1	Multistage Decimator Design Using Goodman–Carey Half-Band Filters	338
5.5	Frequency-Sampling Filters as Bandpass Decimators	341
5.6	Design of Arbitrary Sampling Rate Converters	345
5.6.1	Fractional Delay Rate Change	349
5.6.2	Polynomial Fractional Delay Design	356
5.6.3	B-Spline-Based Fractional Rate Changer	362
5.6.4	MOMS Fractional Rate Changer	366
5.7	Filter Banks	374
5.7.1	Uniform DFT Filter Bank	375
5.7.2	Two-channel Filter Banks	379
5.8	Wavelets	395
5.8.1	The Discrete Wavelet Transformation	398
5.8.2	Discrete Wavelet Transformation Applications	402
	Exercises	411
6.	Fourier Transforms	417
6.1	The Discrete Fourier Transform Algorithms	418
6.1.1	Fourier Transform Approximations Using the DFT	418
6.1.2	Properties of the DFT	420
6.1.3	The Goertzel Algorithm	423
6.1.4	The Bluestein Chirp- z Transform	424
6.1.5	The Rader Algorithm	427
6.1.6	The Winograd DFT Algorithm	434
6.2	The Fast Fourier Transform (FFT) Algorithms	436
6.2.1	The Cooley–Tukey FFT Algorithm	437
6.2.2	The Good–Thomas FFT Algorithm	449
6.2.3	The Winograd FFT Algorithm	451
6.2.4	Comparison of DFT and FFT Algorithms	455
6.2.5	IP Core FFT Design	456
6.3	Fourier-Related Transforms	461
6.3.1	Computing the DCT Using the DFT	462
6.3.2	Fast Direct DCT Implementation	463
	Exercises	465
7.	Communication Systems	475
7.1	Error Control and Cryptography	475
7.1.1	Basic Concepts from Coding Theory	476
7.1.2	Block Codes	482
7.1.3	Convolutional Codes	485
7.1.4	Cryptography Algorithms for FPGAs	494
7.2	Modulation and Demodulation	509
7.2.1	Basic Modulation Concepts	510
7.2.2	Incoherent Demodulation	515
7.2.3	Coherent Demodulation	521

Exercises	529
8. Adaptive Systems	533
8.1 Application of Adaptive Systems	534
8.1.1 Interference Cancellation	534
8.1.2 Prediction	535
8.1.3 Inverse Modeling	535
8.1.4 System Identification	536
8.2 Optimum Estimation Techniques	537
8.2.1 The Optimum Wiener Estimation	540
8.3 The Widrow–Hoff Least Mean Square Algorithm	544
8.3.1 Learning Curves.....	551
8.3.2 Normalized LMS (NLMS)	554
8.4 Transform Domain LMS Algorithms	555
8.4.1 Fast-Convolution Techniques.....	556
8.4.2 Using Orthogonal Transforms	557
8.5 Implementation of the LMS Algorithm	561
8.5.1 Quantization Effects	561
8.5.2 FPGA Design of the LMS Algorithm	561
8.5.3 Pipelined LMS Filters.....	565
8.5.4 Transposed Form LMS Filter	567
8.5.5 Design of DLMS Algorithms	568
8.5.6 LMS Designs using SIGNUM Function	572
8.6 Recursive Least Square Algorithms	575
8.6.1 RLS with Finite Memory	578
8.6.2 Fast RLS Kalman Implementation	581
8.6.3 The Fast a Posteriori Kalman RLS Algorithm.....	586
8.7 Comparison of LMS and RLS Parameters	587
8.8 Principle Component Analysis (PCA)	589
8.8.1 Principle Component Analysis Computation	591
8.8.2 Implementation of Sanger's GHA PCA	595
8.9 Independent Component Analysis (ICA)	601
8.9.1 Whitening and Orthogonalization	603
8.9.2 Independent Component Analysis Algorithm.....	604
8.9.3 Implementation of the EASI ICA Algorithm	605
8.9.4 Alternative BSS Algorithms	610
8.10 Coding of Speech and Audio Signals	612
8.10.1 A- and μ -Law Coding.....	612
8.10.2 Linear and Adaptive PCM Coding.....	617
8.10.3 Coding by Modeling: The LPC-10e Method	623
8.10.4 MPEG Audio Coding Methods.....	624
Exercises	626

9. Microprocessor Design	631
9.1 History of Microprocessors	631
9.1.1 Brief History of General-Purpose Microprocessors	632
9.1.2 Brief History of RISC Microprocessors	634
9.1.3 Brief History of PDSPs	635
9.2 Instruction Set Design	638
9.2.1 Addressing Modes	638
9.2.2 Data Flow: Zero-, One-, Two- or Three-Address Design	646
9.2.3 Register File and Memory Architecture	652
9.2.4 Operation Support	656
9.2.5 Next Operation Location	659
9.3 Software Tools	660
9.3.1 Lexical Analysis	661
9.3.2 Parser Development	672
9.4 FPGA Microprocessor Cores	682
9.4.1 Hardcore Microprocessors	683
9.4.2 Softcore Microprocessors	689
9.5 Case Studies	700
9.5.1 T-RISC Stack Microprocessors	700
9.5.2 LISA Wavelet Processor Design	706
9.5.3 Nios Custom Instruction Design	721
Exercises	728
10. Image and Video Processing	739
10.1 Overview on Image and Video Processing	740
10.1.1 Image Format	741
10.1.2 Basic Image Processing Operation	746
10.2 Case Study 1: Edge Detection in HDL	748
10.2.1 2D HDL Filter Design	751
10.2.2 Imaging System Design	753
10.2.3 Putting the VGA Edge Detection System Together ..	757
10.3 Case Study 2: Median Filter Using an Image Processing Library	769
10.3.1 The Median Filter	770
10.3.2 Median Filter in HDL	772
10.3.3 Nios Median Filtering Image Processing System ..	775
10.3.4 Median Filter in SW	777
10.4 Motion Detection	782
10.4.1 Motion Detection	782
10.4.2 ME Co-processor Design	785
10.4.3 Video Compression Standards	788
Exercises	791
Appendix A. Verilog Code of Design Examples	795

Appendix B. Design Examples Synthesis Results	879
Appendix C. VHDL and Verilog Coding Keywords	883
Appendix D. CD-ROM Content	885
Appendix E. Glossary	895
References	903
Index	923

1. Introduction

This chapter gives an overview of the algorithms and technology we will discuss in the book. It starts with an introduction to digital signal processing and we will then discuss FPGA technology in particular. Finally, the Altera EP4CE115F29C7 and a larger design example, including chip synthesis, timing analysis, floorplan, and power consumption, will be studied.

1.1 Overview of Digital Signal Processing (DSP)

Signal processing has been used to transform or manipulate analog or digital signals for a long time. One of the most frequent applications is obviously the *filtering* of a signal, which will be discussed in Chaps. 3 and 4. Digital signal processing has found many applications, ranging from data communications, speech, audio or biomedical signal processing to instrumentation and robotics. Table 1.1 gives an overview of applications where DSP technology is used [6].

Digital signal processing (DSP) has become a mature technology and has replaced traditional analog signal processing systems in many applications. DSP systems enjoy several advantages, such as insensitivity to change in temperature, aging, or component tolerance. Historically, analog chip design yielded smaller die sizes, but now, with the noise associated with modern submicrometer designs, digital designs can often be much more densely integrated than analog designs. This yields compact, low-power, and low-cost digital designs.

Two events have accelerated DSP development. One is the disclosure by Cooley and Tuckey (1965) of an efficient algorithm to compute the discrete Fourier Transform (DFT). This class of algorithm will be discussed in detail in Chap. 6. The other milestone was the introduction of the programmable digital signal processor (PDSP) in the late 1970s, which will be discussed in Chap. 9. This could compute a (fixed-point) “multiply-and-accumulate” in only one clock cycle, which was an essential improvement compared with the “Von Neuman” microprocessor-based systems in those days. Modern PDSPs may include more sophisticated functions, such as floating-point multipliers, barrelshifters, memory banks, or zero-overhead interfaces to A/D and D/A converters. EDN publishes every year a detailed overview of available PDSPs

Table 1.1. Digital signal processing applications

Area	DSP algorithm
General-purpose	Filtering and convolution, adaptive filtering, detection and correlation, spectral estimation, and Fourier transform
Speech processing	Coding and decoding, encryption and decryption, speech recognition and synthesis, speaker identification, echo cancellation, cochlea-implant signal processing
Audio processing	Hi-fi encoding and decoding, noise cancellation, audio equalization, ambient acoustics emulation, audio mixing and editing, sound synthesis
Image processing	Compression and decompression, rotation, image transmission and decomposing, image recognition, image enhancement, retina-implant signal processing
Information systems	Voice mail, facsimile (fax), modems, cellular telephones, modulators/demodulators, line equalizers, data encryption and decryption, digital communications and LANs, spread-spectrum technology, wireless LANs, radio and television, biomedical signal processing
Control	Servo control, disk control, printer control, engine control, guidance and navigation, vibration control, power-system monitors, robots
Instrumentation	Beamforming, waveform generation, transient analysis, steady-state analysis, scientific instrumentation, radar and sonar

[7]. We will return in Chap. 2 (p. 126) and Chap. 9 to PDSPs after we have studied FPGA architectures.

Figure 1.1 shows a typical application used to implement an analog system by means of a digital signal processing system. The analog input signal is fed through an analog anti-aliasing filter whose stopband starts at half the sampling frequency f_s to suppress unwanted mirror frequencies that occur during the sampling process. Then the analog-to-digital converter (ADC) follows that is typically implemented with a track-and-hold and a quantizer (and encoder) circuit. The digital signal processing circuit then performs the steps that in the past would have been implemented in the analog system. We may want to process further or store (e.g., on CD) the digital processed data, or we may like to produce an analog output signal (e.g., audio signal) via a digital-to-analog converter (DAC) which would be the output of the equivalent analog system.

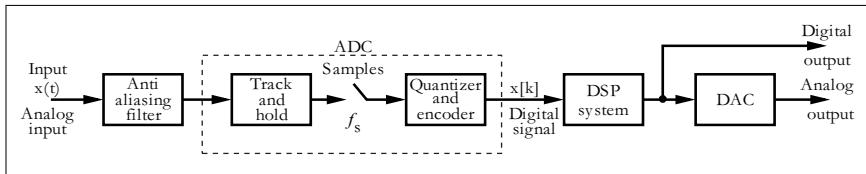


Fig. 1.1. A typical DSP application

1.2 FPGA Technology

VLSI circuits can be classified as shown in Fig. 1.2. FPGAs are a member of a class of devices called field-programmable logic (FPL). FPLs are defined as programmable devices containing repeated fields of small logic blocks and elements². It can be argued that an FPGA is an ASIC technology since FPGAs are application-specific ICs. It is, however, generally assumed that the design of a classic ASIC required additional semiconductor processing steps beyond those required for an FPL. The additional steps provide higher-order ASICs with their performance and power consumption advantage, but also with high nonrecurring engineering (NRE) costs. At 40 nm the NRE costs are about \$4 million; see [8]. Gate arrays, on the other hand, typically consist of a “sea of NAND gates” whose functions are customer provided in a “wire list.” The wire list is used during the fabrication process to achieve the distinct definition of the final metal layer. The designer of a *programmable* gate array solution, however, has full control over the actual design implementation without the need (and delay) for any physical IC fabrication facility. A more detailed FPGA/ASIC comparison can be found in Sect. 1.3, p. 12.

1.2.1 Classification by Granularity

Logic block size correlates to the *granularity* of a device that, in turn, relates to the effort required to complete the wiring between the blocks (routing channels). In general three different granularity classes can be found:

- Fine granularity (Pilkington or “sea of gates” architecture)
- Medium granularity (FPGA)
- Large granularity (CPLD)

Fine-Granularity Devices

Fine-grain devices were first licensed by Plessey and later by Motorola, being supplied by Pilkington Semiconductors. The basic logic cell consisted of a single NAND gate and a latch (see Fig. 1.3). Because it is possible to realize

² Called slice or configurable logic block (CLB) by Xilinx, logic cell (LC), logic element (LE), or adaptive logic module (ALM) by Altera.

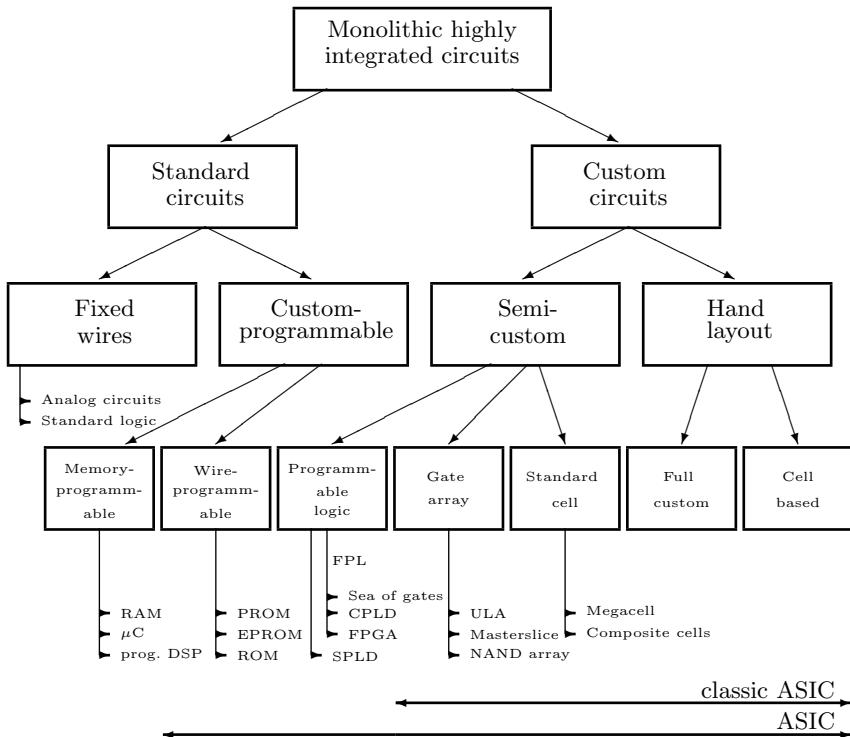


Fig. 1.2. Classification of VLSI circuits

any binary logic function using NAND gates (see Exercise 1.1, p. 47), NAND gates are called *universal* functions. This technique is still in use for gate array designs along with approved logic synthesis tools, such as **ESPRESSO**. Wiring between gate-array NAND gates is accomplished by using additional metal layer(s). For programmable architectures, this becomes a bottleneck because the routing resources used are very high compared with the implemented logic functions. In addition, a high number of NAND gates is needed to build a simple DSP object. A fast 4-bit adder, for example, uses about 130 NAND gates. This makes fine-granularity technologies unattractive in implementing most DSP algorithms.

Medium-Granularity Devices

The most common FPGA architecture is shown in Fig. 1.4a. Concrete examples of contemporary medium-grain FPGA devices are shown in Figs. 1.11 and 1.12 (p. 23). The elementary logic blocks are typically small tables, (typically with 4- to 5-bit input tables, 1- or 2-bit output), or are realized with dedicated multiplexer (MPX) logic such as that used in Actel ACT-2 de-

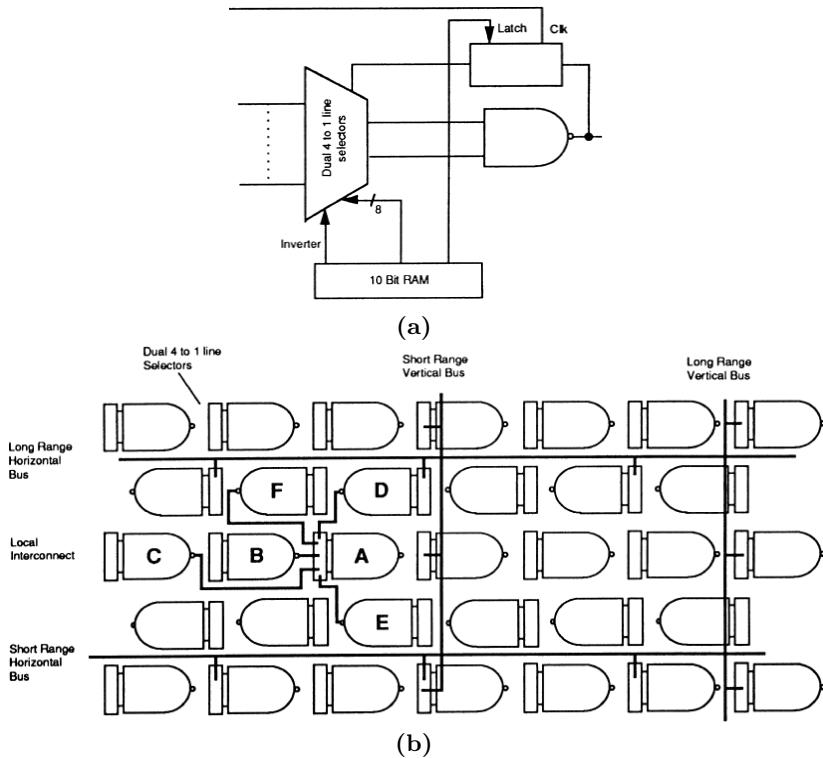


Fig. 1.3. Plessey ERA60100 architecture with 10K NAND logic blocks. **(a)** Elementary logic block. **(b)** Routing architecture. (© Plessey [9])

vices [10]. Routing channel choices range from short to long. A programmable I/O block with flip-flops is attached to the physical boundary of the device.

Large-Granularity Devices

Large granularity devices, such as the complex programmable logic devices (CPLDs), are characterized in Fig. 1.4b. They are defined by combining so-called simple programmable logic devices (SPLDs), like the classic GAL16V8 shown in Fig. 1.5. This SPLD consists of a programmable logic array (PLA) implemented as an AND/OR array and a universal I/O logic block. The SPLDs used in CPLDs typically have 8 to 10 inputs, 3 to 4 outputs, and support around 20 product terms. Between these SPLD blocks wide busses (called programmable interconnect arrays (PIAs) by Altera) with short delays are available. By combining the bus and the fixed SPLD timing, it is possible to provide predictable and short pin-to-pin delays with CPLDs.

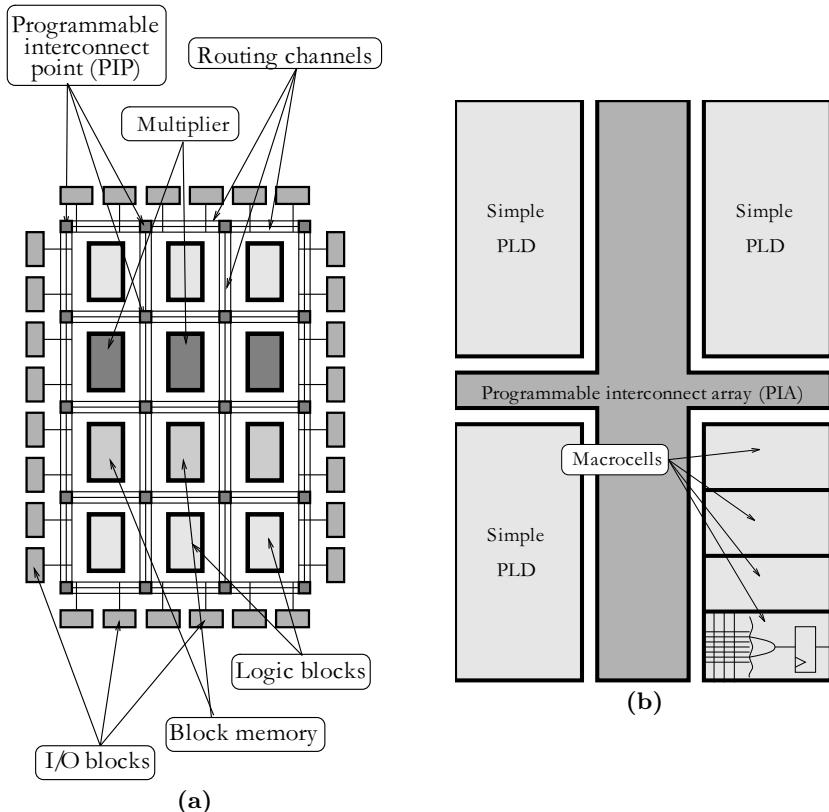


Fig. 1.4. (a) FPGA and (b) CPLD architecture

1.2.2 Classification by Technology

FPLs are available in virtually all memory technologies: SRAM, EPROM, E²PROM, and antifuse [12]. The specific technology defines whether the device is *reprogrammable* or *one-time programmable*. Most SRAM devices can be programmed by a single-bit stream that reduces the wiring requirements, but also increases programming time (typically in the millisecond range). SRAM devices, the dominant technology for FPGAs, are based on static CMOS memory technology, and are re- and in-system programmable. They require, however, an external “boot” device for configuration. Electrically programmable read-only memory (EPROM) devices are usually used in a one-time CMOS programmable mode because of the need to use ultraviolet light for erasure. CMOS electrically erasable programmable read-only memory (E²PROM) can be used as re- and in-system programmable. EPROM and E²PROM have the advantage of a short setup time. Because the programming information is not “downloaded” to the device, it is better protected

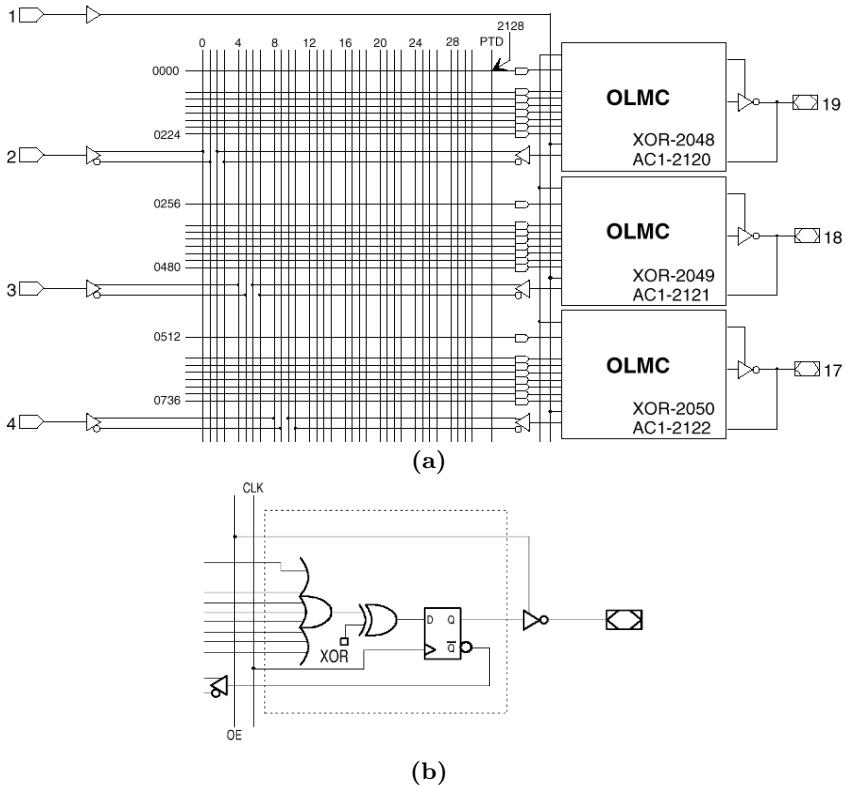


Fig. 1.5. The GAL16V8. (a) First three of eight macrocells. (b) The output logic macrocell (OLMC). (© Lattice [11])

against unauthorized use. A recent innovation, based on an EPROM technology, is called “flash” memory. These devices are usually viewed as “pagewise” in-system reprogrammable systems with physically smaller cells, equivalent to an E²PROM device. Finally, the important advantages and disadvantages of different device technologies are summarized in Table 1.2.

1.2.3 Benchmark for FPLs

Providing objective benchmarks for FPL devices is a nontrivial task. Performance often depends on the experience and skills of the designer, along with design tool features. To establish valid benchmarks, the Programmable Electronic Performance Cooperative (PREP) was founded by Xilinx [13], Altera [14], and Actel [15], and later expanded to more than ten members. PREP has developed nine different benchmarks for FPLs that are summarized in Table 1.3. The central idea underlining the benchmarks is that each

Table 1.2. FPL technology

Technology	SRAM	EPROM	E ² PROM	Antifuse	Flash
Reprogrammable	✓	✓	✓	—	✓
In-system programmable	✓	—	✓	—	✓
Volatile	✓	—	—	—	—
Copy protected	—	✓	✓	✓	✓
Examples	Xilinx Spartan	Altera MAX5K	AMD MACH	Actel ACT	Xilinx XC9500
	Altera Cyclone	Xilinx XC7K	Altera MAX 7K		Cypress Ultra 37K

vendor uses its own devices and software tools to implement the basic blocks as many times as possible in the specified device, while attempting to maximize speed. The number of instantiations of the same logic block within one device is called the *repetition rate* and is the basis for all benchmarks. For DSP comparisons, benchmarks five and six of Table 1.3 are relevant. In Fig. 1.6, repetition rates are reported over frequency, for typical Altera (A_k) and Xilinx (X_k) FPGA and CPLD devices that are currently used on the university development boards. These are not always the largest devices available, but all devices are supported by the web-based version of the design tools. Xilinx seemed to achieve the higher speed, while the Altera FPGAs have a larger number of repetitions. Compared with the CPLDs it can be concluded that modern FPGA families provide the best DSP complexity and maximum speed. This is attributed to the fact that modern devices provide fast-carry logic (see Sect. 1.4.1, p. 20) with delays (less than 0.1 ns per bit) that allow fast adders with large bit width, without the need for expensive “carry look-ahead” decoders. Although PREP benchmarks are useful to compare equivalent gate counts and maximum speeds, for concrete applications additional attributes are also important. They include:

- Array multiplier (e.g., 18×18 bits, 18×25 bits)
- Package such as BGA, TQFP, PGA
- Configuration data stream encryption via DES or AES
- Embedded hardwired microprocessor (e.g., 32-bit ARM Cortex-A9)
- On-chip large block RAM or ROM
- On-chip fast ADC
- External memory support for ZBT, DDR, QDR, SDRAM
- Pin-to-pin delay
- Internal tristate bus

Table 1.3. The PREP benchmarks for FPLs

Number	Benchmark name	Description
1	Data path	Eight 4-to-1 multiplexers drive a parallel-load 8-bit shift register (see Fig. 1.26, p. 48)
2	Timer/counter	Two 8-bit values are clocked through 8-bit value registers and compared (see Fig. 1.27, p. 49)
3	Small state machine	An 8-state machine with 8 inputs and 8 outputs (see Fig. 2.64, p. 174)
4	Large state machine	A 16-state machine with 40 transitions, 8 inputs, and 8 outputs (see Fig. 2.65, p. 176)
5	Arithmetic circuit	A 4-by-4 unsigned multiplier and 8-bit accumulator (see Fig. 4.61, p. 303)
6	16-bit accumulator	A 16-bit accumulator (see Fig. 4.62, p. 303)
7	16-bit counter	Loadable binary up counter (see Fig. 9.42, p. 736)
8	16-bit synchronous prescaled counter	Loadable binary counter with asynchronous reset (see Fig. 9.42, p. 736)
9	Memory mapper	The map decodes a 16-bit address space into 8 ranges (see Fig. 9.43, p. 737)

- Readback- or boundary-scan decoder
- Programmable slew rate or voltage of I/O
- Power dissipation
- Hard IP block for $\times 1$, $\times 2$, or $\times 4$ PCIe

Some of these features are (depending on the specific application) more relevant to DSP application than others. We summarize the availability of some of these key features in Tables 1.4 and 1.5 for Xilinx and Altera, respectively.

The first column shows the device family name. The columns 2 – 9 show the (for most DSP applications) relevant features: (2) the number of address inputs (a.k.a. Fan-in) to the LUT, (3) the size of the embedded array multiplier, (4) the size of the on-chip block RAM measured as kilo (1024) bits, (5) embedded microprocessor: 32-bit ARM Cortex-A9 on current Xilinx ZYNQ and Altera devices, (6) for Xilinx devices the on-chip (Virtex 6: 10 bit, 0.2 MSPS; Series 7: 12 bit 1 MSPS) fast ADC, (7) the target price and availability of the device family. Device that are no longer recommended for

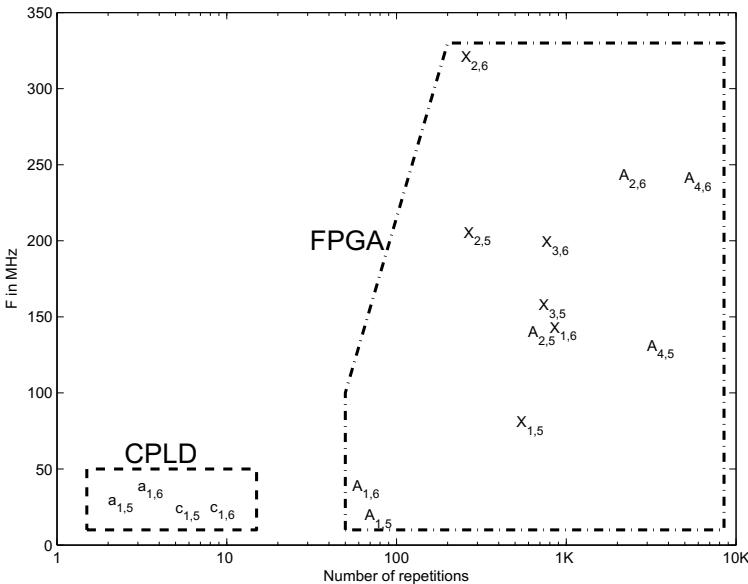


Fig. 1.6. PREP benchmark 5 and 6 (i.e., second subscript) average for FPLs from the Digilent and TERASIC development boards: A₁=FLEX10K from UP2; A₂=Cyclone 2 from DE2; A₃=Cyclone IV from DE2-115; a₁=EPM7128 from UP2; X₁=Spartan 3 from Nexys; X₂=Spartan 6 from Nexys III; X₃=Spartan 6 LX45 from Atlys; and c₁=CoolRunnerII CPLD

new designs are classified as mature with m. Low-cost devices have a single \$ and high price range devices have two \$\$, (8) year the device family was introduced, (9) the process technology used measured in nanometer.

At the time of writing Xilinx supports four device families: the Virtex family for leading performance and capacity, the Kintex family for DSP intensive application and low cost, and the Artix family of lowest cost, replacing the Spartan family of devices. In addition a embedded microprocessor centric family called ZYNQ has been introduced. The Virtex-II, Virtex-4-FX, or Virtex-5-FXT families that included one or more IBM PowerPC RISC processor are no longer recommended for new designs. The Xilinx devices have 18 × 18-bit or 18 × 25-bit embedded multipliers. Most current devices provide an 18 or 36 Kbit memory. An 0.2 MSPS 10 bit fast ADC on-chip has been added for the sixth Virtex generation. The seventh generation includes a 12 bit 1 MSPS dual channel ADC with additional senors for power supply and on-chip temperature with possibly 17 senors sources for the ADCs; see Fig. 1.7b. Keep in mind that a larger number of only the Spartan families are available in the web edition of the development software; most other devices need a subscription edition of the Xilinx ISE software.

Table 1.4. Recent Xilinx FPGA family DSP features

Family	Feature							
	LUT Fan- in	Emb. mult.	BRAM size Kbits	Emb. μP	Fast A/D	Cost / mature	Year	Pro- cess nm
Spartan 3	4	18 × 18	18	—	—	m	2003	90
Spartan 6	6	18 × 18	18	—	—	\$	2009	45
Virtex 4	4	18 × 18	36	PPC	—	m	2004	90
Virtex 5	6	25 × 18	36	PPC	—	m	2006	65
Virtex 6	6	25 × 18	36	—	✓	\$\$	2009	40
Artix 7	6	25 × 18	36	—	✓	\$	2010	28
Kintex 7	6	25 × 18	36	—	✓	\$\$	2010	28
Virtex 7	6	25 × 18	36	—	✓	\$\$	2010	28
ZYNQ-7K	6	25 × 18	36	ARM	✓	\$\$	2011	28

Table 1.5. Altera FPGA family DSP features

Family	Feature							
	LUT Fan- in	Emb. mult. size	BRAM size Kbits	Emb. μP	Fast A/D	Cost ... mature	Year	Pro- cess nm
FLEX10K	4	—	4	—	—	m	1995	420
Cyclone	4	—	4	—	—	\$	2002	130
Cyclone II	4	18 × 18	4	—	—	\$	2004	90
Cyclone III	4	18 × 18	9	—	—	\$	2007	65
Cyclone IV	4	18 × 18	9	—	—	\$	2009	60
Cyclone V	8	27 × 27	10	—	—	\$	2011	28
Arria	8	18 × 18	576	—	—	\$	2007	90
Arria II	8	18 × 18	9	—	—	\$	2009	40
Arria V	8	27 × 27	10	ARM	—	\$\$	2011	28
Stratix	4	18 × 18	0.5,4,512	—	—	\$\$	2002	130
Stratix II	8	18 × 18	0.5,4,512	—	—	\$\$	2004	90
Stratix III	8	18 × 18	9,144	—	—	\$\$	2006	65
Stratix IV	8	18 × 18	9,144	—	—	\$\$	2008	40
Stratix V	8	27 × 27	20	—	—	\$\$	2010	28

Altera offers three main classes of FPGA devices: the Stratix family include high performance devices, the Arria family has the midrange devices, and the Cyclone devices have lowest cost, lowest power, lowest density, and lowest performance of all three. Logic block size of recent devices has been increased from 4 input LUT to a maximum 8 different inputs, that allow for instance to build 3 input adders at almost the same speed as two input adders. Physically the ALM has two flip-flops, two full adders, two 4-input LUTs and four 3-input LUTs, and many multiplexer that allow a general

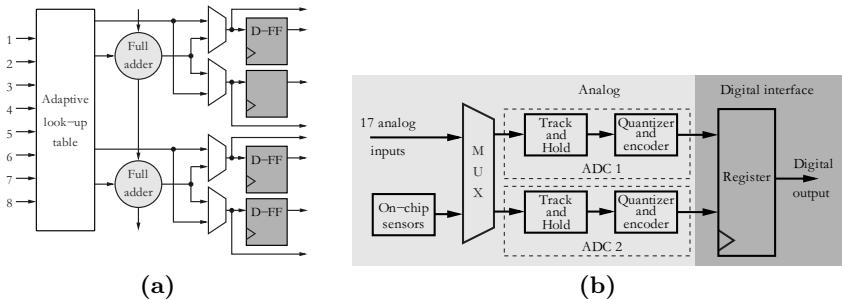


Fig. 1.7. New architecture features used in recent FPGA families. **(a)** Altera’s ALM block. **(b)** Xilinx series 7 high speed on-chip ADC

6-input function to be implemented; see Fig. 1.7a. The embedded multiplier size in Altera devices ranges from 9×9 -bit, 18×18 -bit, to 27×27 -bit. Larger multipliers can be build by grouping this blocks together at the cost of a reduced speed. Starting with the fifth generation three 9-bit blocks are combined into one fast 27×27 -bit multiplier. The memories are available in a wide variety range from 0.5K, M4K, M9K, M10K, M144K, to M512K bit memory. Keep in mind that only the Cyclone family is available in the web 12.1 edition of the Quartus II development software; Arria and Stratix devices need a subscription edition of the software.

Power dissipation of FPL is another important characteristic, in particular in mobil applications. It has been found that CPLDs usually have higher “standby” power consumption. For higher-frequency applications, FPGAs can be expected to have a higher power dissipation. A detailed power analysis example can be found in Sect. 1.4.2, p. 29.

1.3 DSP Technology Requirements

The PLD market share, by vendor, is presented in Fig. 1.8. PLDs, since their introduction in the early 1980s, have enjoyed in the first two decades steady growth of 20% per annum, outperforming ASIC growth by more than 10%. In 2001 the worldwide recession in microelectronics reduced the ASIC and FPLD growth. Since 2003 we see again a steep increase in revenue (growth about 10% per annum) for the two market leaders. Actel became part of Microsemi Inc. in November 2010. The reason that FPLDs outperformed ASICs seems to be related to the fact that FPLDs can offer many of the advantages of ASICs such as:

- Reduction in size, weight, and power dissipation
- Higher throughput
- Better security against unauthorized copies

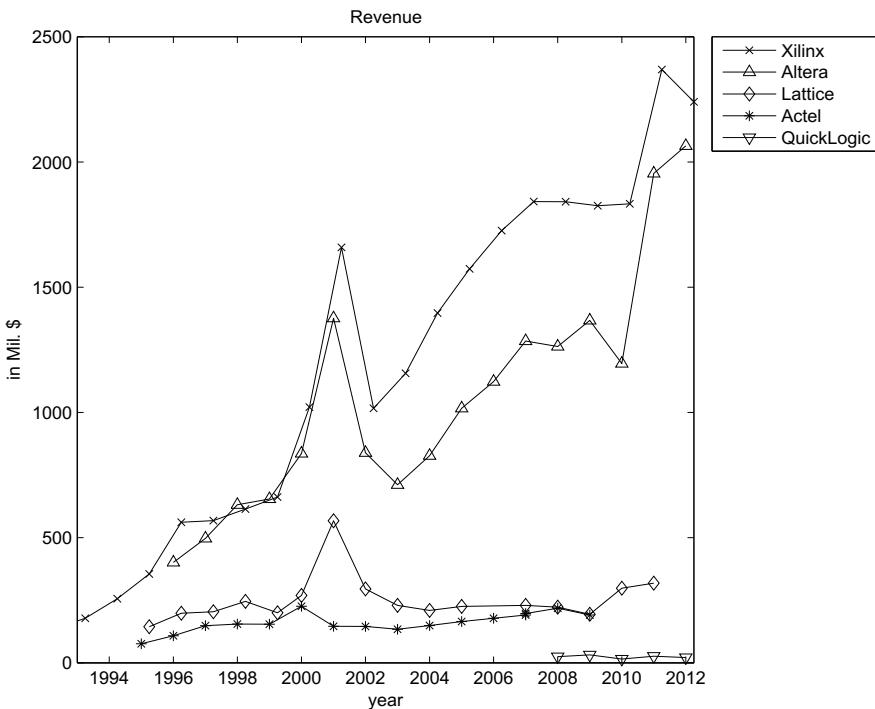


Fig. 1.8. Revenues of the top five vendors in the PLD/FPGA/CPLD market

- Reduced device and inventory costs
- Reduced board test costs

without many of the disadvantages of ASICs such as:

- A reduction in development time (rapid prototyping) by a factor of three to four
- In-circuit reprogrammability
- Lower NRE costs resulting in more economical designs for solutions requiring less than 1000 units

CBIC ASICs are used in high-end, high-volume applications (more than 1000 copies). Compared to FPLs, CBIC ASICs typically have about ten times more gates for the same die size. An attempt to solve the latter problem is the so-called hard-wired FPGA (Altera named HardCopy ASICs and Xilinx now EasyPath FPGAs), where a gate array is used to implement a verified FPGA design.

Table 1.6. Floating-point multiply-accumulate performance comparison of Stratix IV and TI PDSP

Feature	Stratix IV EP4SGX230	TI TMS320C6727B-350
F_{rmax}	227 MHz	350 MHz
# Devices	1	62
Total GFPMACS	43.5	43.4
Total Cost (1K units)	\$871	\$1,799 (62 × \$29.03)

1.3.1 FPGA and Programmable Signal Processors

General-purpose programmable digital signal processors (PDSPs) [6, 16, 17] have enjoyed tremendous success for the last decades. They are based on a reduced instruction set computer (RISC) paradigm with an architecture consisting of at least one fast array multiplier (e.g., 16×16 -bit to 24×24 -bit fixed-point, or 32-bit floating-point), with an extended wordwidth accumulator. The PDSP advantage comes from the fact that most signal processing algorithms are multiply and accumulate (MAC) intensive. By using a multistage pipeline architecture, PDSPs can achieve MAC rates limited only by the speed of the array multiplier. More details on PDSPs can be found in Chap. 9. It can be argued that an FPGA can also be used to implement MAC cells [18], but cost issues will most often give PDSPs an advantage, if the PDSP meets the desired MAC rate. On the other hand we now find many high-bandwidth signal-processing applications such as wireless, multimedia, or satellite transmission, and FPGA technology can provide more bandwidth through multiple MAC cells on one chip. In addition, there are several algorithms such as CORDIC, NTT or error-correction algorithms, which will be discussed later, where FPL technology has proved to be more efficient than a PDSP. It is assumed [19] that in the future PDSPs will dominate applications that require complicated algorithms (e.g., several `if-then-else` constructs), while FPGAs will dominate more front-end (sensor) applications like FIR filters, CORDIC algorithms, or FFTs, which will be the focus of this book.

FPGAs have outperformed fixed-point PDSP in cost and power for many years now. In recent years FPGAs also have improved in the floating-point performance and in particular with a board size, power, and cost standpoint now offering an attractive alternative to a floating-point PDSP solution. For a typical design Table 1.6 lists some key factors. Clearly, on such a large scale design, FPGAs offer substantial improvement compared to a classical floating-point PDSP design. In the example shown in Table 1.6 the same giga floating-point multiply-accumulate operations per second (GFPMACS) are designed using an FPGA and PDSP. We would need 62 floating-point PDSP for the task that can be accomplished with a single FPGA used for the DE4 University boards (DE4 boards are available for ca. \$1000 through the University program; \$3K is the commercial DE4 price). A single FPMAC

Table 1.7. VLSI design levels

Object	Objectives	Example
System	Performance specifications	Computer, disk unit, radar
Chip	Algorithm	μ P, RAM, ROM, UART, parallel port
Register	Data flow	Register, ALU, COUNTER, MUX
Gate	Boolean equations	AND, OR, XOR, FF
Circuit	Differential equations	Transistor, R, L, C
Layout	None	Geometrical shapes

requires about 475 ALMs and 4 embedded 9x9 bit multipliers for the Stratix-IV family [20]. The fastest FP PDSP from TI is the TI320C6727B-350 which is a low-cost high speed floating-point PDSP that runs at 350 MHz and delivers 700 MMACs [21]. We will also observe an substantial board size and power dissipation improvement. The overall device cost is improved by over 100%.

We will discuss floating-point design via IP cores and using the new VHDL 2008 standard (compatible with the VHDL-1993 via a synthesizable library provided by David Bishop) in Sect. 2.7 in more details.

1.4 Design Implementation

The levels of detail commonly used in VLSI designs range from a geometrical layout of full custom ASICs to system design using so-called set-top boxes. Table 1.7 gives a survey. Layout and circuit-level activities are absent from FPGA design efforts because their physical structure is programmable but fixed. The best utilization of a device is typically achieved at the gate level using register transfer design languages. Time-to-market requirements, combined with the rapidly increasing complexity of FPGAs, are forcing a methodology shift towards the use of intellectual property (IP) macrocells or mega-core cells. Macrocells provide the designer with a collection of predefined functions, such as microprocessors or UARTs. The designer, therefore, need only specify selected features and attributes (e.g., accuracy), and a synthesizer will generate a hardware description code or schematic for the resulting solution.

A key point in FPGA technology is, therefore, powerful design tools to:

- Shorten the design cycle
- Provide good utilization of the device
- Provide synthesizer options, i.e., choose between optimization speed vs size of the design

A CAE tool taxonomy, as it applies to FPGA design flow, is presented in Fig. 1.9. The design entry can be graphical or text-based. A formal check

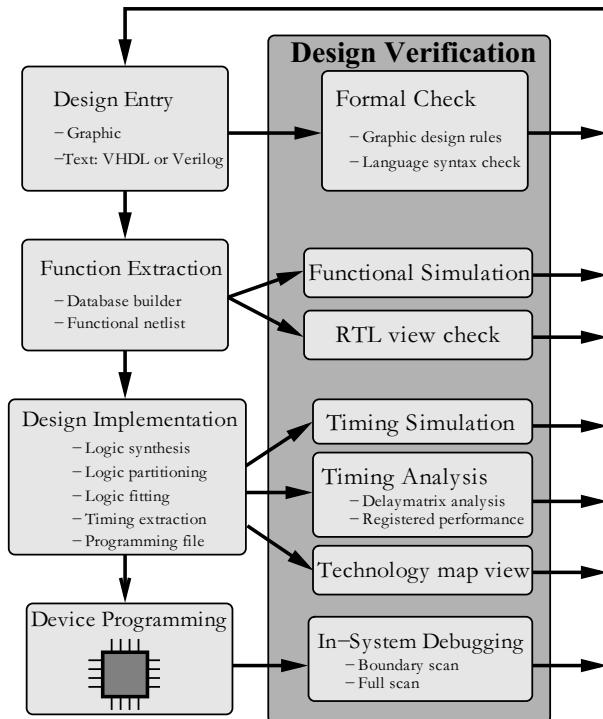


Fig. 1.9. CAD design circle

that eliminates syntax errors or graphic design rule errors (e.g., open-ended wires) should be performed before proceeding to the next step. In the function extraction the basic design information is extracted from the design and written in a functional netlist. The netlist allows a first functional simulation (a.k.a. RTL level simulation) of the circuit and enables the construction of an example data set called a test bench for later testing of the design with timing information. The functional netlist also allows an RTL view of the circuit that gives a quick overview of the circuit described in HDL. If the RTL view verification or functional simulation is not passed we start with the design entry again. If the functional test is satisfactory we proceed with the design implementation, which usually takes several steps and also requires much more compile time than the function extraction. At the end of the design implementation the circuit is completely routed within our FPGA, which provides precise resource data and allows us to perform a simulation with all timing delay information (a.k.a. gate level simulation) as well as performance measurements. Some synthesis tools also offer a technology map view of the circuit that shows how the HDL elements are mapped to LUTs, memory, and embedded multipliers. If all these implementation data are as expected we can proceed with the programming of the actual FPGA; if not we have

to start with the design entry again and make appropriate changes in our design. Using the JTAG interface of modern FPGAs we can also directly monitor data processing on the FPGA: we may read out just the I/O cells (which is called a boundary scan) or we can read back all internal flip-flops (which is called a full scan). If the in-system debugging fails we need to return to the design entry.

In general, the decision of whether to work within a graphical or a text design environment is a matter of personal taste and prior experience. A graphical presentation of a DSP solution can emphasize the highly regular dataflow associated with many DSP algorithms. The textual environment, however, is often preferred with regard to algorithm control design and allows a wider range of design styles, as demonstrated in the following design example. Specifically, for Altera's Quartus II, it seemed that with text design more special attributes and more precise behavior can be assigned in the designs.

Example 1.1: Comparison of VHDL Design Styles

The following design example illustrates three design strategies in a VHDL context. Specifically, the techniques explored are:

- Structural style (component instantiation, i.e., graphical netlist design)
- Data flow, i.e., concurrent statements
- Sequential design using `PROCESS` templates

The VHDL design file `example.vhd`⁴ follows (comments start with `--`):

```

PACKAGE n_bit_int IS      -- User defined type
    SUBTYPE S8 IS INTEGER RANGE -128 TO 127;
END n_bit_int;
LIBRARY work; USE work.n_bit_int.ALL;

LIBRARY ieee;           -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-----
ENTITY example IS          -----> Interface
    GENERIC (WIDTH : INTEGER := 8);   -- Bit width
    PORT (clk   : IN STD_LOGIC;      -- System clock
          reset : IN STD_LOGIC;      -- Asynchronous reset
          a, b, op1 : IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
          sum    : OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0); -- SLV type inputs
          c, d : OUT S8);           -- SLV type output
                                     -- Integer output
END example;
-----
ARCHITECTURE fpga OF example IS
    COMPONENT lib_add_sub
        GENERIC (LPM_WIDTH : INTEGER;
                 LPM_DIRECTION : string := "ADD");

```

⁴ The equivalent Verilog code `example.v` for this example can be found in Appendix A on page 795. Synthesis results are shown in Appendix B on page 881.

```

PORT(dataaa : IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
      datab : IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
      result: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

COMPONENT lib_ff
  GENERIC (LPM_WIDTH : INTEGER);
  PORT (clock : IN STD_LOGIC;
        data : IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
        q     : OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;

SIGNAL a_i, b_i    : S8 := 0;      -- Auxiliary signals
SIGNAL op2, op3   : STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
BEGIN

  -- Conversion int -> logic vector
  op2 <= b;

  add1: lib_add_sub           -----> Component instantiation
    GENERIC MAP (LPM_WIDTH => WIDTH,
                  LPM_DIRECTION => "ADD")
    PORT MAP (dataaa => op1,
              datab => op2,
              result => op3);

  reg1: lib_ff
    GENERIC MAP (LPM_WIDTH => WIDTH )
    PORT MAP (data => op3,
              q => sum,
              clock => clk);

  c <= a_i + b_i;           -----> Data flow style (concurrent)
  a_i <= CONV_INTEGER(a); -- Order of statement does not
  b_i <= CONV_INTEGER(b); -- matter in concurrent code

P1: PROCESS(clk, reset) -----> Behavioral/sequential style
VARIABLE s : S8 := 0;      -- Auxiliary variable
BEGIN
  IF reset = '1' THEN          -- Asynchronous clear
    s := 0; d <= 0;
  ELSIF rising_edge(clk) THEN -- pos. edge triggered FFs
    s := s + a_i;            -----> Sequential statement
    -- d <= s;               -- "d" at this line: b_i would
    s := s + b_i;             -- not be added to output.
    d <= s;                  -- Ordering of statements matters
  END IF;
END PROCESS P1;

END fpga;

```

All HDL code start with the definition of I/O ports followed by the internal nets. In VHDL the library used needs to be specified before the port declarations. Then the actual circuit description starts. The example code shows short coding examples of all three styles: components, concurrent, and

sequential. The `lib_ff` and `lib_add_sub` blocks are components from a component library that are instantiated similar to a graphic design. We need to specify the major parameters and connect the ports of the components to the internal nets or I/O ports of our designs. Next shown is a short sequence of concurrent code. Here the type `SIGNAL` must be used that describes unique nets that can only be used once. However, the order of these statements does not matter. The type conversion does not need to be placed before the arithmetic operation as is necessary in sequential coding (i.e., C-code). Finally, a `PROCESS` example shows the sequential coding style. Here we can also use local type `VARIABLE` that can only be used within the `PROCESS`. This `VARIABLE` does not need to be an unique net in the circuit and can be used (e.g., `VARIABLE s`) multiple times. The ordering of the statements within the `PROCESS` does matter just as in a normal sequential program code. Only the statements up to the assignment are evaluated. For instance, if the assignment to `d` is not at the end of the `IF` statement then `b_i` would not be added to `d`. Another point to note is the strict typing of the VHDL language. The conversion between `INTEGER` and `STD_LOGIC` needs to be done via a function call.

1.1

A detailed study of how to synthesize and simulate a circuit (according to our flow from Fig. 1.9) will follow in Sect. 1.4.3, p. 32. At the end of the CAD tool flow we will have a programming file ready that can be downloaded to a hardware board (like the prototype boards shown in Fig. 1.10). We proceed with programming the device and may perform additional hardware tests using the read-back methods. Altera supports several DSP development boards with a large set of useful prototype components including fast A/D, D/A, audio CODEC, DIP switches, single and seven segment LEDs, and push buttons. These development boards are available from Altera directly. Altera offers Stratix and Cyclone boards, in the \$199-\$24,995 price range, which differ not only in FPGA size but also in terms of the extra features, like number, precision, and speed of A/D channels, and memory blocks. For universities a good choice will be the low-cost DE2-115 Cyclone IV board, which is still more expensive than the UP2 or UP3 boards used in many digital logic labs, but has a two-channel CODEC, large memory bank outside the FPGA, and many other useful ports (USB, VGA, PS/2, Ethernet, seven segment LEDs, LCD, switches, bush buttons, etc.); see Fig. 1.10a. Xilinx on the other hand has very limited direct board support; all boards for instance available in the university program are from third parties. However some of these boards are priced so low that it seems that they are not-for-profit designs. A good board for DSP purposes (with on-chip multipliers and audio CODEC) is the Atlys board offered by Digilent Inc. for only \$199 or \$349, for academic and regular pricing, respectively; see Fig. 1.10b. The board has a Spartan-6 XC6SLX45 FPGA, 16-MByte flash, 128-MByte DDR, eight LEDs, eight switches, and five push buttons. For DSP and video experiments, we can take advantage of the A/D and D/A in the AC-97 audio CODEC and the four HDMI ports.

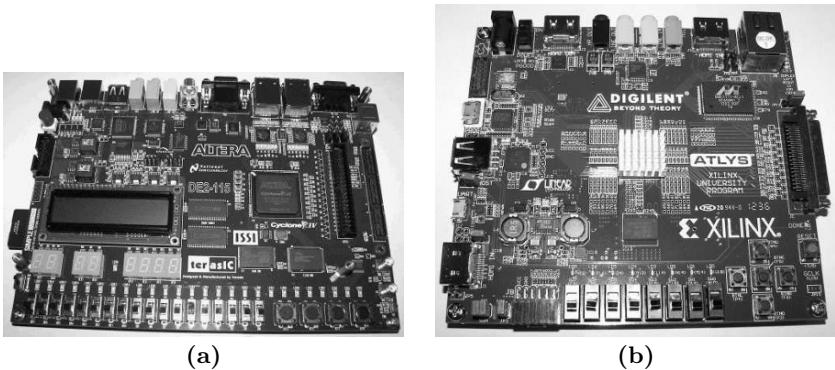


Fig. 1.10. Low-cost prototype boards. (a) Cyclone IV DE2-115 Altera board. (b) Xilinx Atlys with on-board ADC and DAC CODEC

1.4.1 FPGA Structure

At the beginning of the twenty first century FPGA device families now have several attractive features for implementing DSP algorithms. These devices provide fast-carry logic, which allows implementations of 32-bit (non-pipelined) adders at speeds exceeding 300 MHz [1, 22, 23], embedded 18×18 -bit multipliers, and large memory blocks.

Xilinx FPGAs are based on the elementary logic block of the early XC4000 family and the newest derivatives are called Spartan (low cost) and Virtex (high performance). Altera devices are based on FLEX 10K logic blocks and the newest derivatives are called Stratix (high performance) and Cyclone (low cost). The Xilinx devices have the wide range of routing levels typical of a FPGAs, while the Altera devices are based on an architecture with the wide busses used in Altera's CPLDs. However, the basic blocks of the Cyclone and Stratix devices are no longer large PLAs as in CPLD. Instead the devices now have medium granularity, i.e., small look-up tables (LUTs), as is typical for FPGAs. Several of these LUTs, called logic elements (LE) by Altera, are grouped together in a logic array block (LAB). The number of LEs in an LAB depends on the device family, where newer families in general have more LEs per LAB: Flex10K utilizes 8 LEs per LAB, APEX20K uses 10 LEs per LAB, and Cyclone II-IV has 16 LEs per LAB.

Since the Spartan-6 device XC6SLX45 is part of a popular Atlys DSP board offered by Digilent Inc., see Fig. 1.10b, we will have a closer look at this FPGA family. The basic logic elements of the Xilinx Spartan-6 are called slices and come in three different versions: M, L and X. In the Spartan-6 family 2 slices are combined in a configurable logic blocks (CLB), having a total of 8 6-input one-output LUTs (or 16 5-input LUTs), and 16 flip-flops, 256 bit distributed RAM, or 128 bit shift register. Of all slices, 25% are type M and have all these features; 25% slices are of type L and do not have the

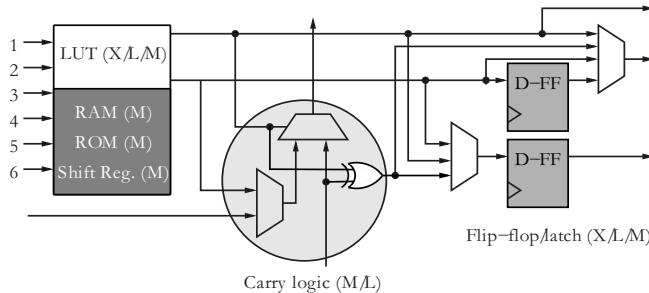


Fig. 1.11. Quarter portion of a Spartan-6 slice. The X slice only has LUT and two flip-flops. The L slice adds the fast carry logic. The M slice has all features. (© Xilinx)

memory shift register function; 50% of the slices are type X and these slices do not have shift register option, arithmetic carry, and wide multiplexer. Figure 1.11 shows one quarter of a slice and the features for each type. Each LUT in the M type slice can be used as a 64×1 RAM or ROM. The Xilinx device has multiple levels of routing, ranging from CLB to CLB, to long lines spanning the entire chip. The Spartan-6 device also includes large memory blocks (18,432 bits or 16,384 bits if no parity bits are used) that can be used as single- or dual-port RAM or ROM. The memory blocks can be configured as $2^9 \times 32, 2^{10} \times 16, \dots, 2^{14} \times 1$, i.e., each additional address bit reduces the data bit width by a factor of two. Another interesting feature for DSP purpose is the embedded multiplier in the Spartan-6 family. These are fast 18×18 -bit signed array multipliers. If unsigned multiplication is required 17×17 -bit multipliers can be implemented with this embedded multiplier. This device family also includes up to four complete clock networks (DCMs) that allow one to implement several designs that run at different clock frequencies (or phases) in the same FPGA with low clock skew. Up to 33 Mbits configuration files size is required to program Spartan-6 devices. Table 1.8 shows the most important DSP features of members of the Xilinx Spartan-6 family.

As an example of an Altera FPGA family let us have a look at the Cyclone IV E device EP4CE115 used in the low-cost prototyping board DE2-115 by Altera; see Fig. 1.10a. The basic block of the Altera Cyclone IV device achieves a medium granularity using small LUTs. The Cyclone device is similar to the Altera 10K device used in the mature UP2 and UP3 boards, with increased RAM blocks memory size to 9 Kbits, which are no longer called EAB as in Flex 10K or ESB as in the APEX family, but rather M9K memory blocks, which better reflects their memory size. The basic logic element in Altera FPGAs is called a logic element (LE)⁵ and consists of a flip-flop, a four-input one-output or three-input one-output LUT and a fast-carry logic,

⁵ Sometimes also called logic cells (LCs) in a design report file.

Table 1.8. The Xilinx Spartan-6 family

Device	Total five-input LUTs	Slices	RAM blocks	CMT 2 DCM 1 PLL	Emb. mult. 18×18	Conf. file Mbit
XC6SLX4	4800	600	12	2	8	2.7
XC6SLX9	11,440	1430	32	2	16	2.7
XC6SLX16	18,224	2278	32	2	32	3.7
XC6SLX25	30,064	3758	52	2	38	6.4
XC6SLX45	54,576	6822	116	4	58	11.9
XC6SLX75	93,296	11,662	172	6	132	19.7
XC6SLX100	126,576	15,822	180	6	180	26.7
XC6SLX150	184,304	23,038	180	6	180	33.9

Table 1.9. Altera's Cyclone IV E device family

Device	Total four-input LUTs	RAM blocks M9K	PLLs/ clock networks	Emb. mul. 18×18	Max. I/O	Conf. file Mbit
EP4CE6	6272	30	2/10	15	179	2.94
EP4CE10	10,320	46	2/10	23	179	2.94
EP4CE15	15,408	56	4/20	56	343	4.09
EP4CE22	22,320	66	4/20	66	153	5.75
EP4CE30	28,848	66	4/20	66	532	9.53
EP4CE40	39,600	126	4/20	116	532	9.53
EP4CE55	55,856	260	4/20	154	374	14.89
EP4CE75	75,408	305	4/20	200	426	19.97
EP4CE115	114,480	432	4/20	266	528	28.57

or AND/OR product term expanders, as shown in Fig. 1.12. Each LE can be used as a four-input LUT in the normal mode, or in the arithmetic mode, as a three-input LUT with an additional fast carry. Sixteen LEs are combined in a logic array block (LAB) in Cyclone IV devices. Each device contains at least one column with embedded 18×18 -bit multipliers and one column M9K memory blocks. One 18×18 -bit multiplier can also be used as two signed 9×9 -bit multipliers. The M9K memory can be configured as $2^8 \times 32$, $2^9 \times 16, \dots, 8192 \times 1$ RAM or ROM. In addition one parity bit per byte is available (e.g., 256×36 configuration), which can be used for data integrity. These M9Ks and LABs are connected through wide high-speed busses as shown in Fig. 1.13. Several PLLs are in use to produce multiple clock domains with low clock skew in the same device. At least 29 Mbits configuration files size is required to program the EP4CE115. Table 1.9 shows the members of the Altera Cyclone IV E family.

If we compare the two routing strategies from Altera and Xilinx we find that both approaches have value: the Xilinx approach with more local and

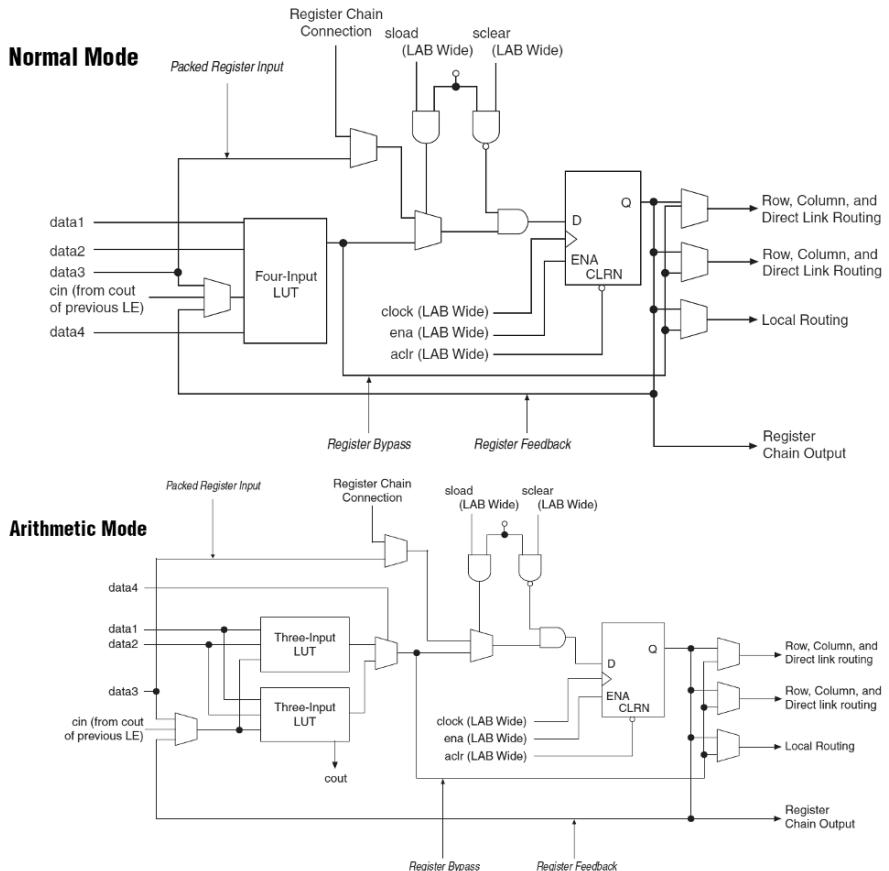


Fig. 1.12. Cyclone IV logic cell. (© Altera [24])

less global routing resources is synergistic to DSP use because most digital signal processing algorithms process the data locally. The Altera approach, with wide busses, also has value, because typically not only are single bits processed in bit slice operations, but also normally wide data vectors with 16 to 32 bits must be moved to the next DSP block.

1.4.2 The Altera EP4CE115F29C7

The Altera EP4CE115F29C7 device, a member of the Cyclone IV E family, which is part of the DSP prototype board DE2-115 provided through Altera's

university program, is used throughout this book. The device nomenclature is interpreted as follows:

```
EP4CE115F29C7
|   |   |
|   |   |||--> Speed grade 7 (lower is faster)
|   |   ||--> Commercial temperature 0-85 Celsius
|   |   |---> Package with 780 pins
|   |   |---> Package FineLine BGA
|   |   |---> LEs in 1000
|-----> Device family: Cyclone IV E
```

Specific design examples will, wherever possible, target the Cyclone IV device EP4CE115F29C7 using Altera-supplied software. The web-based Quartus II software is a fully integrated system with VHDL and Verilog editor, synthesizer, timing evaluations and bitstream generator that can be download free of charge from www.altera.com. The only limitation in the web version is that not all package types and speed grades of every device are supported. Because all examples are available in VHDL and Verilog, any version other than 12.1 or simulator may also be used. For instance, the Xilinx ISE compiler and the ISIM simulator has successfully been used to compile the examples. For other Quartus II software versions the included qvhdl.tcl and qv.tcl can be used to compile all examples for a new software version using just one script.

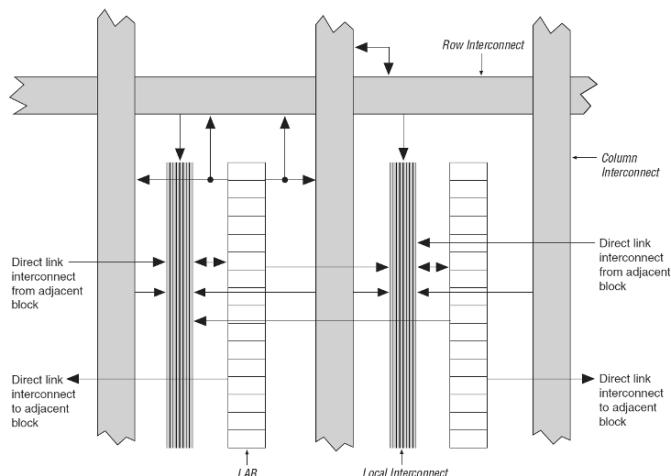


Fig. 1.13. Cyclone IV logic array block resources. (© Altera [24])

Logic Resources

The EP4CE115 is a member of the Altera Cyclone IV family and has a logic density equivalent to about 115,000 logic elements (LEs). An additional 266 multipliers of size 18×18 bits (or twice this number if a size of 9×9 bit is used) are available; see Fig. 1.14. From Table 1.9 it can be seen that the EP4CE115 device has 114,480 basic LEs. This is also the maximum number of implementable full adders. Each LE can be used as a four-input LUT, or in the arithmetic mode as a three-input LUT with an additional fast carry as shown in Fig. 1.12. Sixteen LEs are always combined into a logic array block (LAB); see Fig. 1.13. The number of LABs is therefore $114,480/16 = 7155$. In the left medium area of the device the JTAG interface is placed and uses the area of typically $5 \times 9 = 45$ LABs. This is why the total number of LABs is not just the product of rows \times column. The device also includes six columns of 9-Kbit memory block (called M9K memory blocks) that have the height of one LAB and the total number of M9Ks is therefore $6 \times 72 = 432$. The M9Ks can be configured as 256×36 , 256×32 , 512×18 , 512×16 , ... 8192×1 RAM or ROM, where for each byte one parity bit is available. The EP4CE115 also has four columns of 18×18 bit fast array multipliers, that can also be configured as two 9×9 bit multipliers. The lower left corner of the EP4CE115 is shown in Fig. 1.19 (p. 36) along with the **Bird's Eye View** that shows the overall device floorplan.

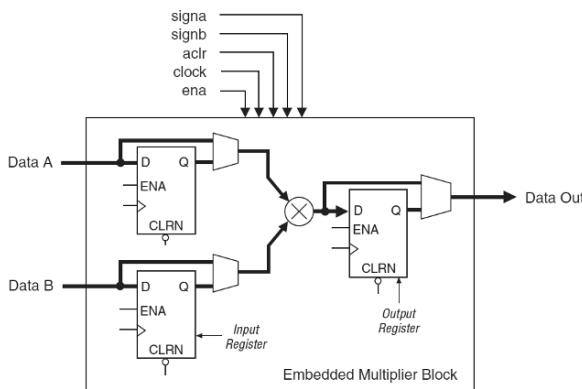


Fig. 1.14. Embedded multiplier architecture. (© Altera [24])

Additional Resources and Routing

While the exact number of horizontal and vertical bus lines for the EP4CE115 is no longer specified in the data sheet as in the older (e.g., FLEX10K) families [25], from the **Compiler Report** we can determine the overall routing re-

Table 1.10. Routing resources for the Altera's Cyclone IV E device EP4CE115

Block	Direct	C4	R4	C16	R24	Clock
342,891	342,891	209,544	289,782	10,120	9963	20

sources of the device; see [Fitter](#)→[Resource Section](#)→[Logic and Routing Section](#). Local connection are provided by block interconnects and direct links. Each LE of the Cyclone IV devices can drive up to 48 LEs through fast local and direct link interconnects. The next level of routing are the R4 and C4 fast row and column local connections that allow wires to reach LABs at a distance of ± 4 LABs, or 3 LABs and one embedded multiplier or M9K memory block. The longest connections for logic available are R24 and C16 wires that allows 24 row or 16 column LAB, respectively. It is also possible to use any combination of row and column connections in case the source and destination LAB are not only in different rows but also in different columns. The global clocks network span the entire FPGA. Table 1.10 gives an overview of the available routing resources in the EP4CE115.

The datasheets provide more details about the available internal LAB control signals and global clocks. The EP4CE115 has four PLLs and a total of 20 clock networks spanning the entire chip to guaranty short clock skew. Each PLL has five output counters that can be used to generated different frequencies or phase offsets. The DRAMs on the DE2 boards for instance require a clock with 0 and -3 ns offset that we will generate with one PLL.

All 16 LEs in a LAB share the same synchronous clear and load signals. Two asynchronous clear, two clocks, and two enable signals are shared by the LEs in the LAB. This will limit the freedom of the control signals within one LAB, but since DSP signals are usually processed in wide bus signals this should not be a problem for our designs.

The LAB local interconnect is driven by row or column bus signals or LEs in the LAB. Neighboring LABs, PLLs, BRAM, embedded multipliers from left or right can also drive the local LAB. As we will see in the next section, the delay in these connections varies widely and the synthesis tool always tries to place logic as close together as possible to minimize the interconnection delay. A 32-bit adder, for instance, would be best placed in two LABs in two rows one above the other; see Fig. 1.19, p. 36.

Timing Estimates

Altera's Quartus II and the Xilinx ISE software in the past had only two timing optimization goals: Area or Speed. However, with increased device density and system-on-chip designs with multiple clock domains, different I/O clock modes, clocks with multiplexers, and clock dividers, this strategy optimizing area or speed for the complete device may not be a good approach

anymore. Altera now offer a more sophisticated timing specification that looks more like a cell-based ASIC design style and is based on the Synopsis Design Constrain (SDC) files. The idea here is that a synthesis tool may have for the same circuit different library elements such as ripple carry, carry save, or fast look-ahead styles for an adder. The tool then first optimizes the design to meet the specified timing constraints and in a second step then optimizes area. If you like similar synthesis results as with previous tool such as minimum area or maximum speed you can achieve this by over- or under-constraining the design. For instance, if you do not specify a SDC file then a 1 GHz target performance is assumed, what is most likely over-constraining your design and device. If you like only to optimize the area, than you should use a very low target clock rate, since then all effort is put in the area optimization. Since the flow based on SDC specification has been recently introduced it may be a good idea to go through some tutorials. The Altera University program offers an introduction in the “Using TimeQuest Timing Analyzer” tutorial. More details are available in the “Quartus II TimeQuest Timing Analyzer Cookbook” with many different multi-clock domain examples.

Since in our design examples we are only interested in optimization for speed, we can use the default setting (i.e., without an SDC file) that over-constrains the design to run at the desired 1 GHz clock rate. We should expect to see a warning message in the compilation report that timing has not been met but that it is intended and should not be a concern.

To achieve optimal performance, it is necessary to understand how the software physically implements the design. It is useful, therefore, to produce a rough estimate of the solution and then determine how the design may be improved.

Example 1.2: Speed of a 32-bit Adder

Assume one is required to implement a 32-bit adder and estimate the design’s maximum speed. The adder can be implemented in two LABs, each using the fast-carry chain. A rough first estimate can be obtained using the carry-in to carry-out delay, which is 66 ps for Cyclone IV speed grade 7. An upper bound for the maximum performance would then be $32 \times 66 \text{ ps} = 2.112 \text{ ns}$ or 473 MHz. But in the actual implementation additional delays occur: first the interconnect delay from the previous register to the first full adder gives an additional delay of 0.501 ns. Next the first carry t_{cgen} must be generated, requiring about 0.414 ns. Finally, at the end of the carry chain the full sum bit needs to be computed (about 536 ps) and the setup time for the output register (87 ps) needs to be taken into account. The results are then stored in the LE register. The following table summarizes these data path timing delays:

LE register clock-to-output delay	t_{co}	=	232 ps
Interconnect delay	t_{ic}	=	501 ps
Data-in to carry-out delay	t_{cgen}	=	414 ps
Carry-in to carry-out delay	$30 \times t_{cico} = 30 \times 66 \text{ ps} = 1980 \text{ ps}$		
LE look-up table delay	t_{LUT}	=	536 ps
LE register setup time	t_{su}	=	87 ps
Total		=	3,802 ps

For possible clock skew an additional 82 ps should be added and the total estimated delay is 3,884 ps, or a rate of 257.47 MHz. The design is expected to use about 32 LEs for the adder and an additional 2×32 to store the input data in the registers (see also Exercise 1.7, p. 48). The **TimeQuest Analyzer** in Fig. 1.15 shows the full timing path including the clock delays. Since we did not specify the desired clock period **TimeQuest** uses the default 1 ns or 1 GHz value, which is in most cases an over-constraint and the **Slack** (see row 7 in Fig. 1.15a) becomes negative as shown in red, i.e., as a violation. To meet timing we would usually take the default 1 ns clock and add the (absolute) negative slack. If we specify a relaxed timing using a **Synopsys Design Constrain** file to 200 MHz (i.e., clock period 5 ns) then the analysis will show that the timing is met by using a positive slack that is displayed in green; see Fig. 1.15b.

1.2

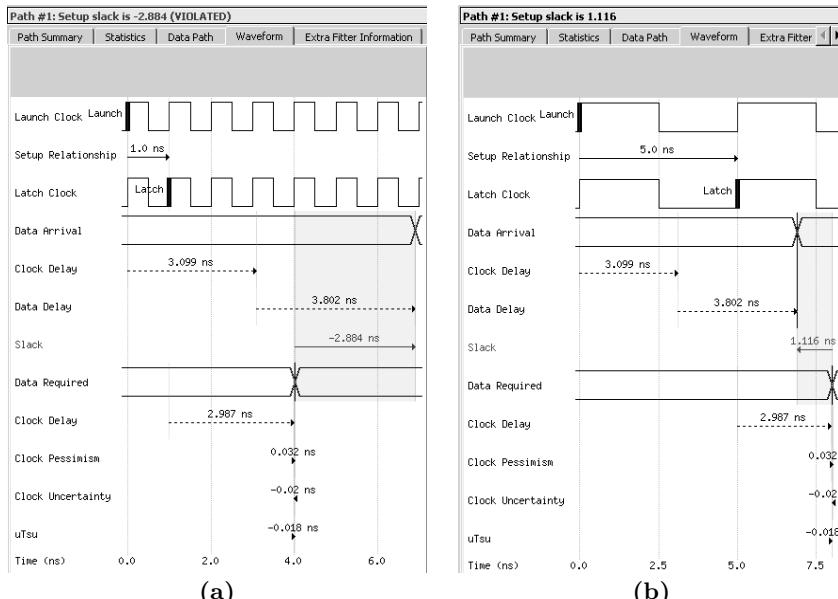


Fig. 1.15. Time Quest analysis. (a) Negative slack shows a timing violation. (b) Positive slack indicating that timing was met

Table 1.11. Some typical registered performance F_{max} and resource data for the Cyclone IV C7 Speed grade

Design	LE	M9K memory blocks	Multiplier blocks 9×9 -bit	Registered Performance MHz
16-bit adder	16(+32)	—	—	363
32-bit adder	32(+64)	—	—	257
64-bit adder	64(+128)	—	—	169
FIFO $2^8 \times 36$	47	1	—	274
RAM $2^8 \times 36$	—	1	—	274
9×9 -bit multiplier	—	—	1	300
18×18 -bit multiplier	—	—	2	250

If the two LABs used cannot be placed in the same column next to each other, then an additional delay would occur. If the signal comes directly from the I/O pins, much longer delays have to be expected. The Quartus II **TimeQuest Analyzer** reports for instance a propagation delay of 11.3 ns for a direct FPGA connection from input `SW[1]` to output pin `HEX0[1]` on the `DE2_115` board. The delays from I/O pins are much larger than the registered performance F_{max} when data comes directly from the register next to the design under test. Datasheets [24, Vol. 3] usually report the best performance that is achieved if I/O data of the design are placed in registers close to the design unit under test. Multiplier and block M9K (but not the adder) have additional I/O registers to enable maximum speed; see Fig. 1.14. The additional I/O registers are usually not counted in the LE resource estimates, since it is assumed that the previous processing unit uses an output register for all data. This may not always be the case and we have therefore put the additional register needed for the adder design in parentheses. Table 1.11 reports some typical data measured under these assumptions. If we compare measured data with the delay given in the data book [24, Vol. 3] we notice that for some blocks the **TimeQuest Analyzer** limits the upper frequency to a specific bound less than the delay in the data book. This is a conservative and more secure estimate – the design may in fact run error free at a slightly higher speed.

Power Dissipation

The power consumption of an FPGA can be a critical design constraint, especially for mobile applications. Using 3.3 V or even lower-voltage process technology devices is recommended in this case. The Cyclone IV family for instance is produced in a 2.5-V transistor, 60-nm low-k dielectric process from the Taiwan ASIC foundry TSMC, but I/O interface voltages of 3.3 V, 2.5 V, 1.8 V, 1.5 V, and 1.2 V are also supported. To estimate the power dissip-

tion of the Altera device EP4CE115, two main sources must be considered, namely:

- 1) Static power dissipation, $P_{\text{static}} \approx 135 \text{ mW}$ for the EP4CE115F29C7 using typical industrial temperature grade
- 2) Dynamic (logic, multiplier, RAM, PLL, clocks, I/O) power dissipation, I_{active}

The first parameter (standby power) in CMOS technology is generally small. The active current depends mainly on the clock frequency and the number of LEs or other resources in use. Altera provides an EXCEL work sheet called **PowerPlay Early Power Estimator** to get an idea about the power consumption (e.g., battery life) and possible cooling requirements in an early project phase.

Table 1.12. Power consumption estimation for the Cyclone IV EP4CE115F29C7

Parameter	Units	Toggle rate (%)	Power mW
P_{static}			159
LEs 114 480 at 100 MHz	114,480	12.5%	1170
M9K block memory	432	50%	74
9 × 9-bit multiplier	532	12.5%	153
I/O cells (2.5 V, 4 mA)	528	12.5%	164
PLL	4		32
Clock network	115,706		486
Total			2238

For LE the dynamic power dissipation is estimated according to the proportional relation

$$P \approx I_{\text{dynamic}} V_{cc} = K \times f_{\text{max}} \times N \times \tau_{\text{LE}} V_{cc}, \quad (1.1)$$

where K is a constant, f_{max} is the operating frequency in MHz, N is the total number of logic cells used in the device, and τ_{LE} is the average percentage of logic cells toggling at each clock (typically 12.5%). Table 1.12 shows the results for power estimation when all resources of the EP4CE115F29C7 are in use and a system clock of 100 MHz is applied. For less resource usage or lower system clock the data in (1.1) can be adjusted. If, for instance, a system clock is reduced from 100 MHz to 10 MHz then the power would be reduced to $159 + 2079/10 = 366.9 \text{ mW}$, and the static power consumption would now account for 43%.

Although the **PowerPlay Estimation** is a useful tool in a project planning phase, it has its limitations in accuracy because the designer has to specify the toggle rate. There are cases when it becomes more complicated, such

as in frequency synthesis design examples; see Fig. 1.16. While the block RAM estimation with a 50% toggle may be accurate, the toggle rate of the LEs in the accumulator part is more difficult to determine, since the LSBs will toggle at a much higher frequency than the MSBs, since the accumulators produce a triangular output function. A more accurate power estimation can be made using Altera's **PowerPlay Power Analyzer Tool** available from the **Processing** menu. The **Analyzer** allows us to read in toggle data computed from the simulation output. The simulator produces a "Signal Activity File" or "Value Change Dump" file that can be selected as the input file for the **Analyzer**. The **PowerPlay Power Analyzer** tool allows different options that we can select. We can just specify a clock rate via the **TimeQuest Timing Analyzer** SDC file and set a default toggle rate, without a simulator's help. More accuracy should be expected if we use a functional or RTL simulation as input for the power estimation that only requires one full compilation of the design. The best estimation should be achieved if we use the compiled design in the MODELSIM simulator input that also includes precise LE switching, glitches, bus driver data, etc. Table 1.13 shows a comparison between the power estimation and the three power analyzer.

Table 1.13. Power consumption for the design shown in Fig. 1.16 for a Cyclone IV EP4CE115F29C7 at 50 MHz using SDC TimeQuest file and Excel-based PowerPlay Early Power Estimator (PPEPE) or Quartus-based PowerPlay Power Analyzer (PPPA). The estimated toggle percentage is 12.5%

	PPEPE Estimation	PPPA Estimation	PPPA RTL Sim.	PPPA Timing
VCD needed	–	–	✓	✓
Parameter	power/mW	power/mW	power/mW	power/mW
Static	135	98.4	98.4	98.5
Dynamic	2	2.3	2.6	3.7
I/O	4	38.9	38.9	50.4
Total	141	139.6	139.9	152.6

We notice a discrepancy of 10% between early estimation and analysis using timing information. The analysis however requires a complete design including a reliable test bench, while the estimation may be done at an early phase in the project.

The following case study should be used as a detailed scheme for the examples and self-study problems in subsequent chapters.

1.4.3 Case Study: Frequency Synthesizer

The design objective in the following case study is to implement a classical frequency synthesizer based on the Philips PM5190 model (circa 1979; see Fig. 1.16). The synthesizer consists of a 32-bit accumulator, with the eight most significant bits (MSBs) wired to a sine ROM lookup table (LUT) to produce the desired output waveform. The equivalent HDL text file `fun_text.v` and `fun_text.vhd` implement the design using behavioral HDL code, thus avoiding LPM component instantiations. The challenge of the design is to have behavioral HDL code that is synthesized by Altera Quartus II and Xilinx ISE software and work well with the recommended simulators (MODEL-SIM for Altera and ISIM for Xilinx) for both, RTL and timing simulation. In the following we walk through all steps that are usually performed when implementing a design using Quartus II:

- 1) Compilation of the design
- 2) Design results and floor plan
- 3) Simulation of the design
- 4) A performance evaluation

Design Compilation

To check and compile the file, start the Quartus II Software and select **File**→**Open Project** or launch **File**→**New Project Wizard** if you do not have a project file yet. In the project wizard specify the project directory you would like to use, and the project name and top-level design as `fun_text`. Then press **Next** and specify the HDL file you would like to add, in our case

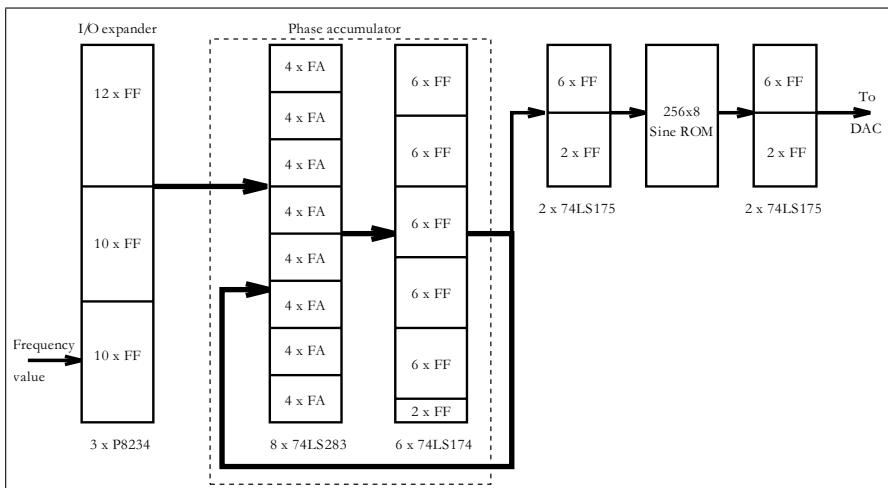


Fig. 1.16. PM5190 frequency synthesizer

`fun_text.vhd`. Press **Next** again and then select the device EP4CE115F29C7 from the Cyclone IV E family (5. from bottom in device listing in Quartus 12.1). Click **Next** and then select MODELSIM-ALTERA as Simulation tool and press **Finish**. If you use the project file from the CD the file `fun_text.qsf` will already have the correct file and device specification. Now select **File** → **Open** to load the HDL file. The VHDL design⁶ reads as follows:

```
-- A 32 bit function generator using accumulator and ROM
LIBRARY ieee;
USE ieee.STD_LOGIC_1164.ALL;
USE ieee.STD_LOGIC_arith.ALL;
USE ieee.STD_LOGIC_signed.ALL;
-- -----
ENTITY fun_text IS
    GENERIC ( WIDTH      : INTEGER := 32);      -- Bit width
    PORT (clk      : IN STD_LOGIC; -- System clock
          reset   : IN STD_LOGIC; -- Asynchronous reset
          M       : IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
          -- Accumulator increment
          acc     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
          -- Accumulator MSBs
          sin     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
    END fun_text;                                -- System sine output
-- -----
ARCHITECTURE fpga OF fun_text IS

COMPONENT sine256x8
    PORT (clk : IN STD_LOGIC;
          addr : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
          data : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
    END COMPONENT;

SIGNAL acc32 : STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
SIGNAL msbs  : STD_LOGIC_VECTOR(7 DOWNTO 0);
          -- Auxiliary vectors
BEGIN

PROCESS (reset, clk, acc32)
BEGIN
    IF reset = '1' THEN
        acc32 <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
        acc32 <= acc32 + M; -- Add M to acc32 and

```

⁶ The equivalent Verilog code `fun.text.v` for this example can be found in Appendix A on page 796. Synthesis results are shown in Appendix B on page 881.

```

END IF;                                -- store in register

END PROCESS;

msbs <= acc32(31 DOWNTO 24); -- Select MSBs
acc <= msbs;

-- Instantiate the ROM
ROM: sine256x8 PORT MAP
      (clk => clk, addr => msbs, data => sin);

END fpga;

```

The object **LIBRARY**, found early in the code, contains predefined modules and definitions. The **ENTITY** block specifies the I/O ports of the device and generic variables. Next in the coding comes the component in use and additional **SIGNAL** definitions. The HDL coding starts after the keyword **BEGIN**. The first **PROCESS** includes the 32-bit accumulator, i.e., an adder followed by a register. The accumulator has an asynchronous active high reset. The next two statements connect the local signal to I/O ports. Finally the ROM table is instantiated as a component and the port signals of the component are connected to the local signal within our design. You may want to have a look at the ROM table design file **sine256x8.vhd**. You can load the file or you can double click it in the Project Navigator window (top left); see Fig. 1.17a. In general a synthesizable ROM or RAM design with initial data loaded in the memory is not a trivial task. A good starting point is either to have a look in the VHDL 1076.6-2004 subset of VHDL for synthesizable code or a little easier is to have a look at the language templates suggest by the tool vendors (Altera: **Edit**→**Insert Template**→**VHDL**→**Full Designs**→**RAMs and ROMs**→**Dual-Port ROM**. or Xilinx: **Edit**→**Language Template**→**VHDL**→**Synthesis Constructs**→**Coding Examples**→**ROM**→**Example Code**). Altera recommend use of a function call for the initialization, and Xilinx a **CONSTANT** array initial definition. It turns out that the latter approach works well with both tools as well as with the simulators for functional and timing simulators and will be the preferred method for the rest of the book. The synthesis attributes as defined in VHDL 1076.6-2004 are currently not supported by either vendors. In VERILOG RAM and ROM initialization is already specified in the language reference manual (LRM) and the most reliable method is to use the **\$readmemh()** function in combination with an **initial** statement.

To optimize the design for speed, go to **Assignment** → **Settings** → **Analysis & Synthesis Settings**. Under **Optimization Technique** click on **Speed**. Timing requirement can be set using the SDC file. The default speed is set to 1 ns that usually works well with the **Speed** optimization. Now start the compiler tool (it's the thick right arrow symbol) that can be found

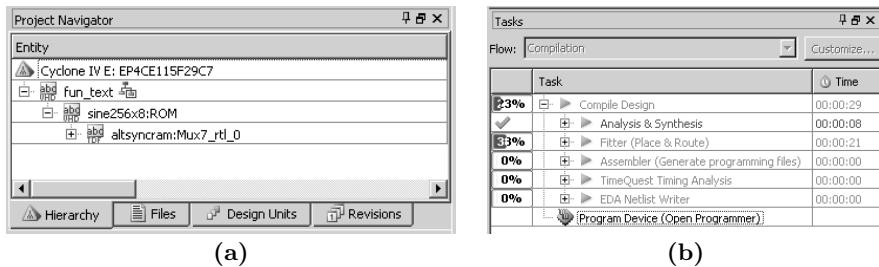


Fig. 1.17. (a) Project Navigator. (b) Compilation steps in Quartus II

under the **Processing** menu. A window left to our HDL window as shown in Fig. 1.17b will show the progress in the compilation. You can see all the steps involved in the compilation, namely: **Analysis & Synthesis**, **Fitter**, **Assembler**, **Timing Analysis**, **Netlist Writer**, and **Program Device**. Alternatively, you can just start the **Analysis & Synthesis** by clicking on **Processing**→**Start**→**Start Analysis & Synthesis** or with **<Ctrl K>**. The compiler checks for basic syntax errors and produces a report file that lists resource estimation for the design. After the syntax check is successful, compilation can be started by pressing the large **Start** button or by pressing **<Ctrl L>**. If all compiler steps were successfully completed, the design is fully implemented. Press the **Compilation Report** button (IC symbol with a paper) from the top menu buttons and the flow summary report should show 32 LEs and 2048 memory bits used. Check the memory initialization file **sine256x8.vhd** for VHDL and **sine256x8.txt** for Verilog. These files were generated using the program **sine.exe** included on the CD-ROM under **util**. Figure 1.17b summarizes all the processing steps of the compilation, as shown in the Quartus II compiler window.

For a *graphical* verification that the HDL design describes the desired circuit, using Altera's Quartus II software, we can use the RTL viewer. The results for the **fun_text.vhd** circuit is shown in Fig. 1.18. To start the RTL viewer click on **Tools**→**Netlist Viewer**→**RTL Viewer**. The other netlist viewer called **Technology Map Viewer** gives a precise picture of how the circuit is mapped onto the FPGA resources. However, to verify the HDL code even for a small design as the function generator the technology map provides too much detail to be helpful, and we will not use it in our design studies.

Floor Planning

The design results can be verified by clicking on the sixth button (i.e., **Chip Planner** or opening the **Tool**→**Chip Planner**) to get a more detailed view of the chip layout. The **Chip Planner** view is shown in Fig. 1.19. Use the **Zoom in** button (i.e., the ± magnifying glass) to produce the screen shown in

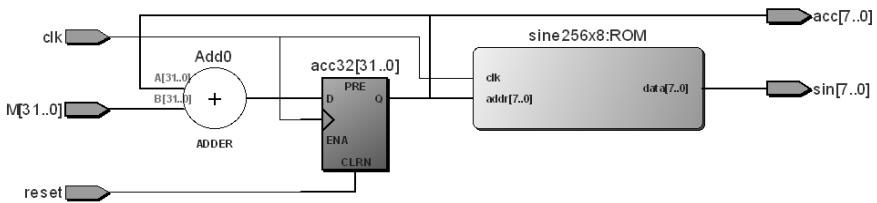


Fig. 1.18. RTL view of the frequency synthesizer

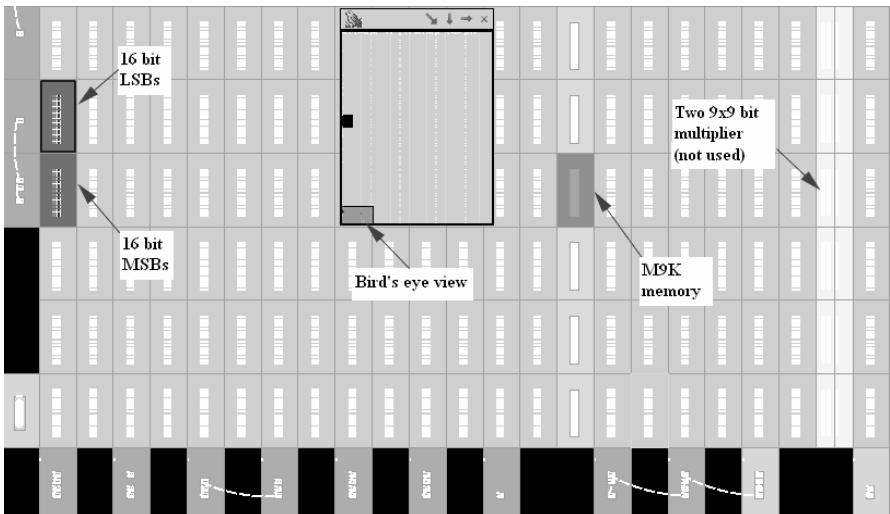


Fig. 1.19. Floorplan of the frequency synthesizer design

Fig. 1.19. Zoom in to the area where the LAB and an M9K are highlighted in a different color. You should then see the two LABs used by the accumulator highlighted in blue and the M9K block highlighted in green. In addition, several I/O cells are also highlighted in brown. Click on the **Bird's Eye View** button⁷ on the left menu buttons and an additional window will pop up. Now select for instance the M9K block and then press the button **Generate Fan-In Connections** or **Generate Fan-Out Connections** several times and more and more connections will be displayed.

Simulation

In terms of simulator we have seen the two FPGA market leaders take opposite directions in recent years. In the past Altera favored the internal VWF

⁷ Note, as with all MS Windows programs, just move your mouse over a button (no need to click on the button) and its name/function will be displayed.

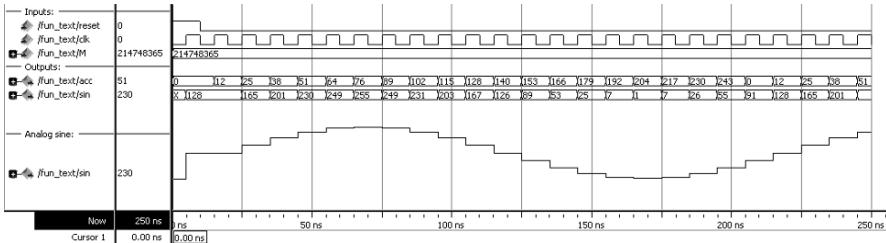


Fig. 1.20. MODELSIM RTL simulation of frequency synthesizer design

simulator (up to Quartus II version 9.1) and now recommends the external MODELSIM-ALTERA or Qsim. Xilinx on the other hand, since version 12.3 (end of 2010), no longer provides a free MODELSIM simulator and instead provides a free embedded ISIM simulator that is integrated within the ISE tool.

When simulating a design such as the ISIM simulator we have the option to use a stimuli file from a TCL script similar to MODELSIM DO files, or we can write a test bench in HDL. A test bench is a short HDL file, where we instantiate the circuit to be tested and then generate and apply our test signals with a statement like

```
clk <= NOT clk AFTER 5 ns;
```

to generate a clock with a $2 \times 5 \text{ ns} = 10 \text{ ns}$ clock period. However, the difficulty with the Xilinx test bench comes from the fact that the circuit with timing information (i.e., *_timesim.vhd) is synthesized directly from the netlist and the STD_LOGIC is used throughout the whole ENTITY description. The original ENTITY data types and GENERIC variables are ignored. If we like to use the same VHDL test bench for RTL and timing simulation then the ENTITY will be restricted to a single data type. More precise, we cannot use INTEGER, SIGNED or FLOAT data types, and even BUFFER or GENERIC parameter would not be permitted. This would therefore interfere greatly with the coding for design reuse and we would need to use a separate test bench for RTL and timing simulation. However, if we do not use a test bench and simulate our circuit directly using the TCL stimuli script, then we can use the same script for RTL and timing simulation. Furthermore, for VHDL and Verilog the same stimuli file can be used; only the compile sequence will be different. The ISIM TCL scripts and MODELSIM DO files are also very similar in their coding style to simplify a transition between the two simulators.

The Altera Quartus II software comes with two free simulator options. The MODELSIM-ALTERA allows us to use the professional tool from Mentor Graphic Inc. The second alternative is the Altera Qsim tool that may have a few less feature than MODELSIM (e.g., no analog waveform) but is also a little easier to handle since it does not require one to write HDL test benches or DO file scripts to assign I/O signals. However, at the time of writing

the `Qsim` in 12.1 does not support the Cyclone IV devices and we therefore select the MODELSIM-ALTERA as default simulator. Moving between VHDL and Verilog stimuli file and Altera and Xilinx is also simplified when using MODELSIM-ALTERA DO files and not HDL test benches.

Before we discuss a simulation example let us first have a look at the file names used, where * stands for the project name. For the RTL simulation we use the *.vhd and *.v files. For timing simulation four different files names are used: *.vho and *.vo for VHDL and Verilog using Altera tools and *_timesim.vhd and *_timesim.v for Xilinx. For both RTL and timing simulation we use the same stimuli files *.do (Altera) and *.tc1 (Xilinx).

To simulate, open the MODELSIM-ALTERA tool. You should see that many predefined libraries are already loaded. Use `File → Change Directory` to move to the directory that includes the HDL files and the simulation scripts. Use `dir *.do` and `dir *.vhd` to verify that the files are indeed in the current path. To run the script type `do fun_text.do 0` for RTL and `do fun_text.do 1` for timing simulation. For functional simulation you should get a result similar to Fig. 1.20. For timing simulation you need first to compile the design in Quartus II or ISE and then simulate the HDL code with timing information. The script compiles the files, opens a waveform and simulates the circuit. The do script for this example looks as follows:

```

set project_name "fun_text"
do tb_ini.do $1 sine256x8

##### Add I/O signals to wave window
add wave -divider "Inputs:"
add wave reset clk
radix -unsigned
add wave M
add wave -divider "Outputs:"
add wave acc sin
add wave -divider -height 80 {Analog sine:}
add wave -color Red -format Analog-Step \
    -radix unsigned -scale 0.25 sin

##### Add stimuli data
force clk 0 0 ns, 1 5 ns -r 10 ns
force reset 1 0 ns, 0 10 ns
force M 214748365 0 ns

##### Run the simulation
run 250 ns
wave zoomfull
configure wave -gridperiod 5ns
configure wave -timelineunits ns

```

The simulation script has typically four parts:

- 1) First the project name is defined and all components in the project are compile via `vcom` or `vlog` for VHDL and Verilog, respectively. The top level project is compiled in a second brief script call to `tb_ini.do` with one parameter (0 for RTL and 1 for timing simulation). It also includes a function to add local signals. These local signals may not be available in the timing simulation.
- 2) Next signals are added to the wave window in the order they should appear in the wave window. We start with the input signals, followed by the outputs. Divider are used to help with differentiations or to add extra information. Note in particular the last entry that shows how an “analog” display of a signal can be defined.
- 3) Next follows in any order the stimuli data to the wave form signals we have just defined. We can define periodic signals as for the `clk` and non-periodic as for `reset` and `M`.
- 4) Finally we run the simulation for a specific time and zoom to display the full time frame. A grid and time unit can also be defined for the wave window.

As can be seen from the script or the wave window the following data have been used. As the period $1/100\text{ MHz} = 10\text{ ns}$ was selected, we set $M = 214,748,364 = 2^{32}/20$, so that the period of the synthesizer is 20 clock cycles long. Note that the ROM has been coded in binary offset (i.e., zero = 128). This is a typical coding used in D/A converters. A short MATLAB script or C-program can be used to generate the data. The CD includes a short C-program `sine.exe` that generates the VHDL code for the component as well as the table data for the Verilog code. For brevity hex display has been used but the program `sine.exe` also allows binary or octal.

Performance Analysis

In order to display timing data a full compile of the design has to be done first. For the Altera tool we use the default setting with period of 1 ns equivalent to 1 GHz clock frequency that will appear as `FMAX_REQUIREMENT "1 ns"` in Quartus II QSF files. Since our design most likely runs slower this will ensure the Quartus compiler is synthesizing the fastest design possible. On the downside we will always get a compiler warning that `Synopsys Design Constraints File 'fun.text.sdc'` was not found and a `Critical Warning: Timing requirements not met`, but we can ignore these messages.

The result of the `TimeQuest` analysis is provided in the `Compilation Report` shown in Fig. 1.21. It includes the three sets of timing data for the slow 85C, 0C, and fast 0C models. We use the most pessimistic, i.e., the slow 85C model. The `Fmax Summary` frequency is the maximum frequency our circuit can run with. Sometime the performance is further restricted due

to the maximum I/O pin speed of 250 MHz. If there is no register-to-register path in the circuit as in a pure combinational design this **Fmax** will come up empty with the message **No paths to report**.

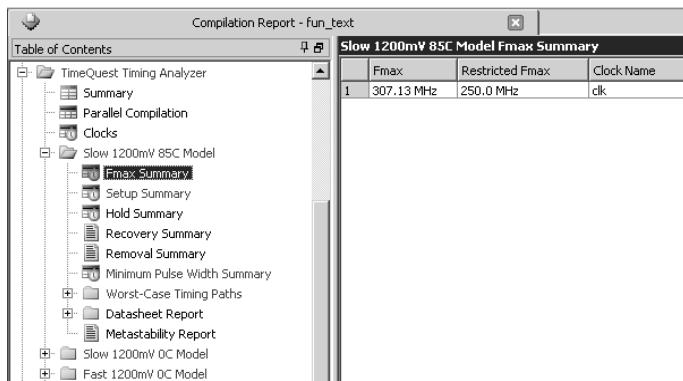


Fig. 1.21. Registered performance of frequency synthesizer design from the **TimeQuest Timing Analyzer**

The performance analysis with the Xilinx software is a little simpler since we have just the two settings for **Speed** and **Area**. In the Implementation view just right click the **Synthesize - XST** entry and under **Synthesis Options** select for the **Optimization Goal** the entry **Speed**. After a full compile check the **Post-PAR Static Timing Report** and you will find the maximum clock as last timing entry in the report as **Clock to Setup on destination clock clk**.

This concludes the case study of the frequency synthesizer.

1.4.4 Design with Intellectual Property Cores

Although FPGAs are known for their capability to support rapid prototyping, this only applies if the HDL design is already available and sufficiently tested. A complex block like a PCI bus interface, a pipelined FFT, an FIR filter, or a μ P may take weeks or even months in development time. One option that allows us essentially to shorten the development time is available with the use of a so-called intellectual property (IP) core. These are predeveloped (larger) blocks, where typical standard blocks like numeric controlled oscillators (NCO), FIR filters, FFTs, or microprocessors are available from FPGA vendors directly, while more specialized blocks (e.g., AES, DES, or JPEG codec, I2C bus or ethernet interfaces) are available from third-party vendors. While some blocks are free in the Quartus II package the larger more sophisticated blocks may have a high price tag. However, as long as the block meets your design requirement it is most often more cost effective to use one of these predefined IP blocks.

Let us now have a quick look at different types of IP blocks and discuss the advantages and disadvantages of each type [26–28]. Typically IP cores are divided into three main forms, as described below.

Soft Core

A *soft core* is a behavioral description of a component that needs to be synthesized with FPGA vendor tools. The block is typically provided in a hardware description language (HDL) like VHDL or Verilog, which allows easy modification by the user, or even new features to be added or deleted before synthesis for a specific vendor or device. On the downside the IP block may also require more work to meet the desired size/speed/power requirements. Very few of the blocks provided by FPGA vendors are available in this form, like the first generation Nios microprocessor from Altera or the PICO blaze microprocessor by Xilinx. IP protection for the FPGA vendor is difficult to achieve since the block is provided as synthesizable HDL and can quite easily be used with a competing FPGA tool/device set or a cell-based ASIC. The price of third-party FPGA blocks provided in HDL is usually much higher than the moderate pricing of the parameterized core discussed next.

Parameterized Core

A *parameterized* or firm core is a structural description of a component. The parameters of the design can be changed before synthesis, but the HDL is usually not available. The majority of cores provided by Altera and Xilinx come in this type of core. They allow certain flexibility, but prohibit the use of the core with other FPGA vendors or ASIC foundries and therefore offer better IP protection for the FPGA vendors than soft cores. Examples of parameterized cores available from Altera and Xilinx include an NCO, FIR filter compiler, FFT (parallel and serial), and embedded processors, e.g., Nios II from Altera. Another advantage of parameterized cores is that usually a resource estimation (LE, multiplier, block RAMs) is available that is most often correct within a few percent, which allows a fast design space exploration in terms of size/speed/power requirements even before synthesis. Test benches in HDL (for MODELSIM simulator) that allow cycle-accurate modeling as well as C or MATLAB scripts that allow behavior-accurate modeling are also standard for parameterized cores. Code generation usually only takes a few seconds. Later in this section we will study an NCO parameterized core and continue this in later chapters (Chap. 3 on FIR filter and Chap. 6 on FFTs).

Hard Core

A *hard core* (fixed netlist core) is a physical description, provided in any of a variety of physical layout formats like EDIF. The cores are usually optimized

for a specific device (family), when hard realtime constraints are required, like for instance a PCI bus interface. The parameters of the design are fixed, like a 16-bit 256-point FFT, but a behavior HDL description allows simulation and integration in a larger project. Most third-party IP cores from FPGA vendors and several free FFT cores from Xilinx use this core type. Since the layout is fixed, the timing and resource data provided are precise and do not depend on synthesis results. However, the downside is that a parameter change is not possible, so if the FFT should have 12- or 24-bit input data the 16-bit 256-point FFT block cannot be used.

IP Core Comparison and Challenges

If we now compare the different IP block types we have to choose between design flexibility (soft core) and fast results and reliability of data (hard core). Soft cores are flexible, e.g., change of system parameters or device/process technology is easy, but may have longer debug time. Hard cores are verified in silicon. Hard cores reduce development, test, and debug time but no VHDL code is available to look at. A parameterized core is most often the best compromise between flexibility and reliability of the generated core.

There are however two major challenges with current IP block technology, which are pricing of a block and, closely related, IP protection. Because the cores are reusable, vendor pricing has to rely on the number of units of IP blocks the customer will use. This is a problem known for many years in patent rights and most often requires long licence agreements and high penalties in case of customer misuse. FPGA-vendor-provided parameterized blocks (as well as the design tool) have very moderate pricing since the vendor will profit if a customer uses the IP block in many devices and then usually has to buy the devices from this single source. This is different with third-party IP block providers that do not have this second stream of income. Here the licence agreement, especially for a soft core, has been drafted very carefully.

For the protection of parameterized cores the FPGA vendor use FlexLM-based keys to enable/disable single IP core generation. Evaluation of the parameterized cores is possible down to hardware verification by using time-limited programming files or requiring a permanent connection between the host PC and board via a JTAG cable, allowing you to program devices and verify your design in hardware before purchasing a licence. For instance, Altera's OpenCore evaluation feature allows you to simulate the behavior of an IP core function within the targeted system, verify the functionality of the design, and evaluate its size and speed quickly and easily. When you are completely satisfied with the IP core function and you would like to take the design into production, you can purchase a licence that allows you to generate non-time-limited programming files. The Quartus software automatically downloads the latest IP cores from Altera's website. Many third-party IP providers also support the OpenCore evaluation flow but you have to contact the IP provider directly in order to enable the OpenCore feature.

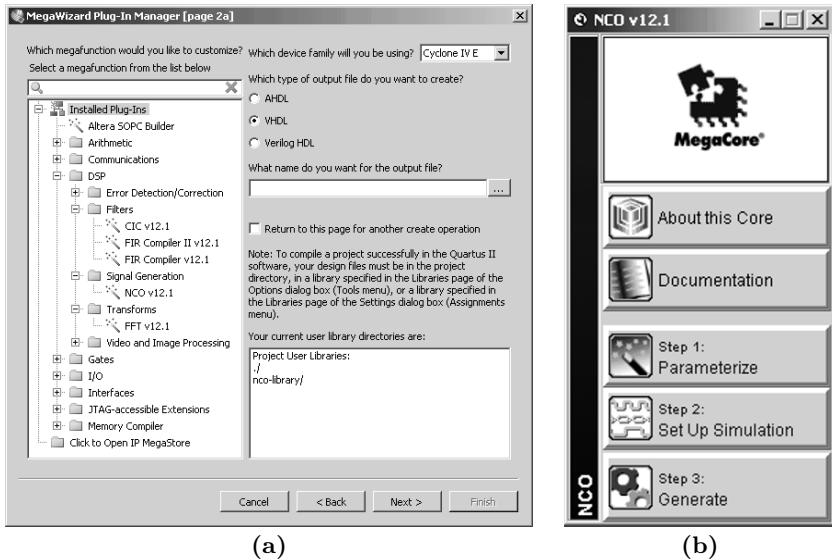


Fig. 1.22. IP design of NCO. (a) Library element selection. (b) IP toolbench

The protection of soft cores is more difficult. Modification of the HDL to make them very hard to read or embedding watermarks in the high-level design by minimizing the extra hardware have been suggested [28]. The watermark should be robust, i.e., a single bit change in the watermark should not be possible without corrupting the authentication of the owner.

IP Core-Based NCO Design

Finally we evaluate the design process of an IP block in an example using the case study from the last section, but this time our design will use Altera's NCO core generator. The NCO compiler generates numerically controlled oscillators (NCOs) optimized for Altera devices. You can use the IP toolbench interface to implement a variety of NCO architectures, including ROM-based, CORDIC-based, and multiplier-based options. The **MegaWizard** also includes time- and frequency-domain graphs that dynamically display the functionality of the NCO based on the parameter settings. For a simple evaluation we name the IP core as our project, i.e., NCO. Open a new project and name it NCO and press the button **Tools**→**MegaWizard Plug-In Manager**. In the first step select the NCO block in the **MegaWizard Plug-In Manager** window; see Fig. 1.22a. The NCO block can be found in the **Signal Generation** group under the DSP cores. We then select the desired output format (AHDL, VHDL, or Verilog) and specify our working directory. Then the IP toolbench pops up (see Fig. 1.22b) and we have access to documentation and can start with step 1, i.e., the parameterization of the block. Since

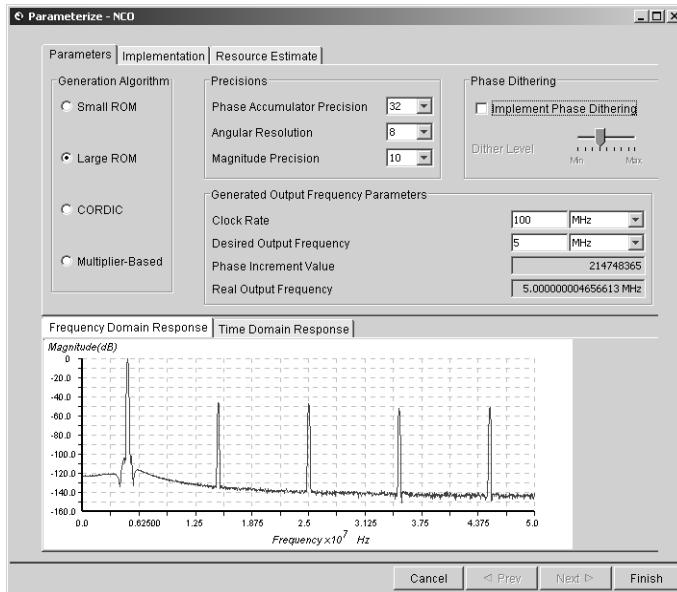


Fig. 1.23. IP parameterization of NCO core according to the data from the `fun_text` case study in the previous section

we want to reproduce the function generator from the last section, we select a 32-bit accumulator. However, the smallest output bit width is 10 bits, and we will not be able to use the 8 bit as in our previous design. We use the **Large ROM** generation algorithm in the parameter window; see Fig. 1.23. The clock rate in the `fun_text` study was 100 MHz and the output period had 20 clock cycles or equivalent 5 MHz frequency. Phase dithering will make the noise more equally distributed, but will require more than twice as many LEs. With phase dithering we would get about 60 dB sidelobe suppression; without dithering we get about 50 dB sidelobe suppression, as can be seen in the **Frequency Domain Response** plot in the lower part of the NCO window. In the **Implementation** window we select **Single Output** since we only require one sine but no cosine output as is typical for I/Q receivers; see Chap. 7. The **Resource Estimation** provides as data 72 LEs, 2560 memory bits, or one M9K block. After we are satisfied with our parameter selection we then proceed to step 2 to specify whether we want to generate behavior HDL code, which speeds up simulation time. Since our block is small we deselect this option and use the full HDL generated code directly. We can now continue with step 3, i.e., **Generate** on the Toolbench. The listing in Table 1.14 gives an overview of the generated files.

We see that not only are the VHDL files generated along with their component file, but MATLAB (bit accurate) and MODELTECH (cycle accurate) test benches are also provided to enable an easy verification path. We decide

Table 1.14. IP file generation for the NCO core

File	Description
<code>nco.vhd</code>	VHDL top-level description of the custom IP core function
<code>nco.cmp</code>	VHDL component declaration for the IP core function variation
<code>nco.bsf</code>	Quartus II symbol file for the IP core function variation
<code>nco_st.v</code>	Generated NCO synthesizable netlist
<code>nco.vho</code>	VHDL IP functional simulation model
<code>nco_tb.vhd</code>	VHDL test bench
<code>nco_vho_msim.tcl</code>	MODEL SIM TCL Script to run the VHDL IP functional simulation model in the MODEL SIM simulation software
<code>nco_wave.do</code>	MODEL SIM waveform file
<code>nco_model.m</code>	MATLAB M-file describing a MATLAB bit-accurate model
<code>nco_tb.m</code>	MATLAB test bench
<code>nco_sin.hex</code>	Intel Hex-format ROM initialization file
<code>nco.vec</code>	Quartus vector file
<code>nco_nativelink.tcl</code>	NativeLink simulation test bench
<code>nco.qip</code>	Quartus project information
<code>nco.html</code>	IP core function report file that lists all generated files

to use the core directly as our top level design entry, therefore avoiding the need to instantiate our block in another design and connect the input and outputs. By inspecting the top level VHDL file `nco.vhd` we notice that the ENTITY has the expect block inputs `clk`, `phi_inc_i`, and output `fsin_o` signals, but has some additional useful control signal, i.e., `reset_n`, `clken`, and `out_valid`, whose function is self-explanatory. We then start a full compilation to generate the `nco.vho` file for the timing simulation. With the full compile data available we can now compare the actual resource requirement with the estimate. The memory requirement and block RAM predictions were correct, but for the LEs with 88 LEs (actual) to 72 LEs (estimated) we observe an 18% error margin.

To simulate the design we use the generated TCL script. Start the MODEL SIM simulator and change to the NCO directory and then type in the command window do `nco_vho_msim.tcl`. Several libraries and the design files

are compiled and a 22,000 ns long simulation is performed. We zoom in to the first couple of clock cycles to see the output valid delay (\approx 6 clock cycles) and the sine period, as shown in Fig. 1.24. The same phase increment value $M = 214,748,365$ as in our function generator (see Fig. 1.20, p. 37) is used and we get a period of 20 clock cycles in the output signal. We may notice a small problem with the IP block, since the output is a signed value, but our D/A converter expects unsigned (or more precisely binary offset) numbers. In a soft core we would be able to change the HDL code of the design, but in the parameterized core we do not have this option. However, we can solve this problem by attaching an adder with constant 512 to the output that makes it an offset binary representation. The offset binary is not a parameter we could select in the block, and we encounter extra design effort. This is a typical experience with the parameterized cores – the cores provide a 90% or more reduction in design time, but sometimes small extra design effort is necessary to meet the exact project requirements.

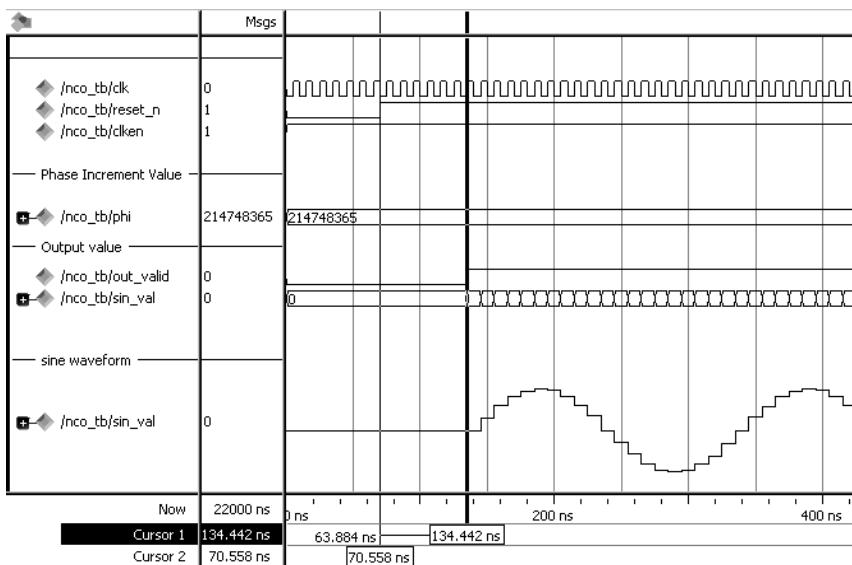


Fig. 1.24. Test bench for NCO IP design. Verification via timing simulation

Exercises

Note: If you have no prior experience with the Quartus II software, refer to the case study found in Sect. 1.4.3, p. 32. If not otherwise noted use the EP4CE115F29C7 from the Cyclone IV E family for the Quartus II synthesis

evaluations.

1.1: Use only two input NAND gates to implement a full adder:

(a) $s = a \oplus b \oplus c_{in}$

(Note: \oplus =XOR)

(b) $c_{out} = a \times b + c_{in} \times (a + b)$

(Note: \times =OR; \times =AND)

(c) Show that the two-input NAND is *universal* by implementing NOT, AND, and OR with NAND gates.

(d) Repeat (a)-(c) for the two input NOR gate.

(e) Repeat (a)-(c) for the two input multiplexer $f = xs' + ys$.

1.2: (a) Compile the HDL file `example` using the MODELSIM-ALTERA tool (see p. 17). Start the MODELSIM-ALTERA tool and change to the directory with the HDL file. Generate a work directory with `vlib work`. Then start the compilation with `vcom example.vhd`.

(b) Simulate the design using the file `example.do`. Start the script with `do example.do 0` for a functional simulation.

(c) Compile the HDL file `example` using the Quartus II compiler with **Timing**. Perform a full compilation using the **Compiler Tool** under the **Processing** menu.

(d) Simulate the design with timing using the script `example.do`. Start the script with `do example.do 1` for a timing simulation.

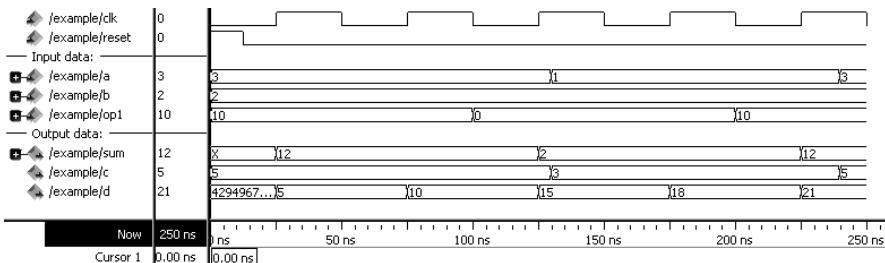


Fig. 1.25. Waveform file for Example 1.1 on p. 17

1.3: (a) Write a MODELSIM-ALTERA simulation script `example.do` and match the stimuli for `clk,a,b,op1` that approximates that shown in Fig. 1.25.

(b) Conduct a simulation using the script `example.do`.

(c) Explain the algebraic relation between `a,b,op1` and `sum,d`.

1.4: (a) Compile the HDL file `fun_text` with the synthesis optimization technique set to **Speed**, **Balanced** or **Area** that can be found in the **Analysis & Synthesis Settings** under **EDA Tool Settings** in the **Assignments** menu.

(b) Evaluate registered performance **Fmax** for Slow 85C timing model and the LE's utilization of the designs from (a). Explain the results.

1.5: Develop a SDC file for the design as shown in the Altera tutorial **Using TimeQuest Timing Analyzer**. Compile the HDL file `fun_text` with the synthesis Optimization Technique set to **Speed** that can be found in the **Analysis &**

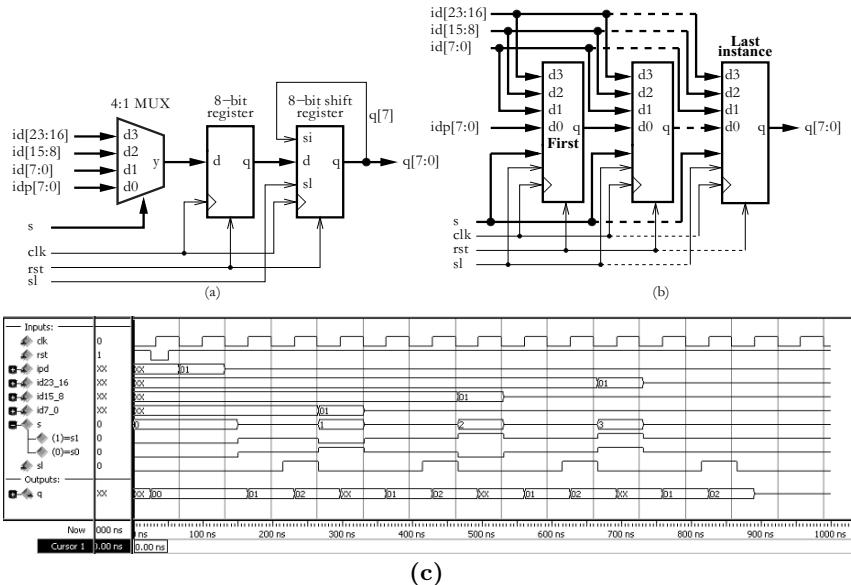


Fig. 1.26. PREP benchmark 1. (a) Single design. (b) Multiple instantiation. (c) Test bench to check the function

Synthesis Settings under EDA Tool Settings in the Assignments menu.
For the period of the clock signal

- (a) 20 ns,
- (b) 10 ns,
- (c) 5 ns,
- (d) 3 ns,

use the clock report to determine the slack.

Note that for each clock period specification you need to run a complete recompile of the circuit. Also report any change in the resource of the circuits.

- 1.6:** (a) Modify the file `fun_text.vhd` to use a `lpm_rom` component and a MIF file `sine.mif`. Simulate the circuit in MODELSIM-ALTERA.
 (b) In Quartus select `File→Open` to open the file `sine.mif` and the file will be displayed in the `Memory editor`. Now select `File→Save As` and select `Save as type: (*.hex)` to store the file in Intel HEX format as `sine.hex`.
 (c) Change the `fun_text` HDL file so that it uses the Intel HEX file `sine.hex` for the ROM table, and verify the correct results through a simulation.

- 1.7:** (a) Design a 32-bit adder using the the Quartus II software.
 (b) Add I/O register and measure the registered performance `Fmax`. Compare the result with the data from Example 1.2 (p. 27).

- 1.8:** (a) Design the PREP benchmark 1, as shown in Fig. 1.26a with the Quartus II software. PREP benchmark no. 1 is a data path circuit with a 4-to-1 8-bit multiplexer, an 8-bit register, followed by a shift register that is controlled by a shift/load input `sl`. For `sl=1` the contents of the register is cyclic rotated by one

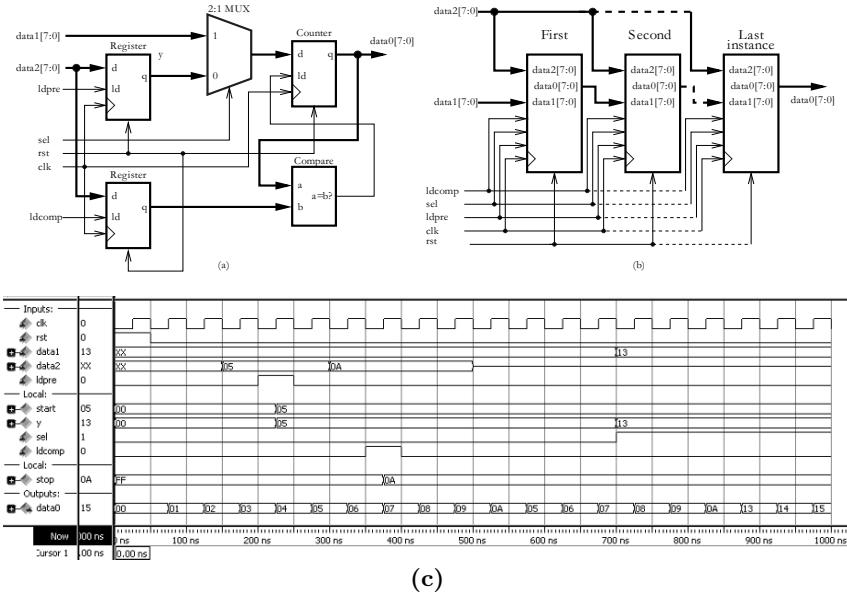


Fig. 1.27. PREP benchmark 2. (a) Single design. (b) Multiple instantiation. (c) Test bench to check the function

bit, i.e., $q(k) = q(k - 1)$, $1 \leq k \leq 7$ and $q(0) \leq q(7)$. The reset **rst** for all flip-flops is an asynchronous reset and the 8-bit registers are positive-edge triggered via **clk**, see the simulation in Fig. 1.26c for the function test.

(b) Determine the registered performance **Fmax** using the **TimeQuest** slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for a single copy. Compile the HDL file with the synthesis **Optimization Technique** set to **Speed**, **Balanced** or **Area**; this can be found in the **Analysis & Synthesis Settings** section under **EDA Tool Settings** in the **Assignments** menu. Which synthesis options are optimal in terms of size and registered performance **Fmax**?

Select one of the following devices:

(b1) EP4CE115F29C7 from the Cyclone IV E family

(b2) EP2C35F672C6 from the Cyclone II family

(b3) EPM7128SLC84-7 from the MAX7000S family

(c) Design the multiple instantiation for benchmark 1 as shown in Fig. 1.26b.

(d) Determine the registered performance **Fmax** using the **TimeQuest** slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for the design with the maximum number of instantiations of PREP benchmark 1. Use the optimal synthesis option you found in (b) for the following devices:

(d1) EP4CE115F29C7 from the Cyclone IV E family

(d2) EP2C35F672C6 from the Cyclone II family

(d3) EPM7128SLC84-7 from the MAX7000S family

1.9: (a) Design the PREP benchmark 2, as shown in Fig. 1.27a with the Quartus II software. PREP benchmark no. 2 is a counter circuit where 2 registers are

loaded with start and stop values of the counter. The design has two 8-bit register and a counter with asynchronous reset `rst` and synchronous load enable signal (`ld`, `ldpre` and `ldcomp`) and positive-edge triggered flip-flops via `clk`. The counter can be loaded through a 2:1 multiplexer (controlled by the `sel` input) directly from the `data1` input or from the register that holds `data2` values. The load signal of the counter is enabled by the equal condition that compares the counter value `data` with the stored values in the `ldcomp` register. Try to match the simulation in Fig. 1.27c for the function test. Note there is a mismatch between the original PREP definition and the actual implementation: We cannot satisfy, that the counter start counting after reset, because all register are set to zero and `ld` will be true all the time, forcing counter to zero. Also in the simulation test bench signal value have been reduced that simulation fits in a 1 μ s time frame.

(b) Determine the registered performance `Fmax` using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for a single copy. Compile the HDL file with the synthesis Optimization Technique set to Speed, Balanced or Area; this can be found in the Analysis & Synthesis Settings section under EDA Tool Settings in the Assignments menu. Which synthesis options are optimal in terms of size and registered performance `Fmax`?

Select one of the following devices:

(b1) EP4CE115F29C7 from the Cyclone IV family

(b2) EP2C35F672C6 from the Cyclone II family

(b3) EPM7128SLC84-7 from the MAX7000S family

(c) Design the multiple instantiation for benchmark 2 as shown in Fig. 1.27b.

(d) Determine the registered performance `Fmax` using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for the design with the maximum number of instantiations of PREP benchmark 2. Use the optimal synthesis option you found in (b) for the following devices:

(d1) EP4CE115F29C7 from the Cyclone IV E family

(d2) EP2C35F672C6 from the Cyclone II family

(d3) EPM7128SLC84-7 from the MAX7000S family

1.10: Use the Quartus II software and write two different codes using the structural (use only one or two input basic gates, i.e., NOT, AND, and OR) and behavioral HDL styles for:

(a) A 2:1 multiplexer

(b) An XNOR gate

(c) A half-adder

(d) A 2:4 decoder (demultiplexer)

Note for VHDL designs: use the `a_74xx` Altera SSI component for the structural design files. Because a component identifier cannot start with a number Altera has added the `a_` in front of each 74 series component. In order to find the names and data types for input and output ports you need to check the library file `libraries\vhdl\altera\maxplus2.vhd` in the Altera Quartus II installation path. You will find that the library uses `STD_LOGIC` data type and the names for the ports are `a_1`, `a_2`, and `a_3` (if needed).

(e) Verify the function of the design(s) via

(e1) A Functional simulation.

(e2) The RTL Viewer that can be found under the Netlist Viewers in the Tools menu.

1.11: Use the Quartus II software language templates and compile the HDL designs for:

(a) A tri-state buffer, see Logic→Tri-State

(b) A flip-flop with all control signals, see Logic→Registers→Full-Featured

Positive Edge Register with All Secondary Signals

- (c) A binary counter, see **Full Designs**→**Arithmetic**→**Counters**
 (d) A Moore State Machine with asynchronous reset, see **Full Designs**→**State Machines**

Open a new HDL text file and then select **Insert Template** from the **Edit** menu.

- (e) Verify the function of the design(s) via
 (e1) A **Functional simulation**
 (e2) The **RTL Viewer** that can be found under the **Netlist Viewers** in the **Tools** menu

1.12: Use the **search** option in Quartus II software help to study HDL designs for:

- (a) The 14 counters, see **search**→**implementing sequential logic**
 (b) A manually specifying state assignments, **Search**→**enumsmch**
 (c) A latch, **Search**→**latchinf**
 (d) A one's counter, **Search**→**proc**→**Using Process Statements**
 (e) A implementing CAM, RAM & ROM, **Search**→**ram256x8**
 (f) A implementing a user-defined component, **Search**→**reg24**
 (g) Implementing registers with clr, load, and preset, **Search**→**reginf**
 (h) A state machine, **Search**→**state_machine**→**Implementing...**

Open a new project and HDL text file. Then **Copy/Paste** the HDL code, save and compile the code. Note that in VHDL you need to add the **STD_LOGIC_1164** IEEE library so that the code runs error free.

- (i) Verify the function of the design via
 (i1) A **Functional simulation**
 (i2) The **RTL Viewer** that can be found under the **Netlist Viewers** in the **Tools** menu

1.13: Determine if the following VHDL identifiers are valid (true) or invalid (false).

- (a) VHSIC (b) h333 (c) A_B_C
 (d) XyZ (e) N#3 (f) My-name
 (g) BEGIN (h) A_-B (i) ENTITI

1.14: Determine if the following VHDL string literals are valid (true) or invalid (false).

- (a) B"11_00" (b) 0"5678" (c) 0"0_1_2"
 (d) X"5678" (e) 16#FfF# (f) 10#007#
 (g) 5#12345# (h) 2#0001_1111_# (i) 2#00_00#

1.15: Determine the number of bits necessary to represent the following integer numbers.

- (a) INTEGER RANGE 10 TO 20;
 (b) INTEGER RANGE -2**6 TO 2**4-1;
 (c) INTEGER RANGE -10 TO -5;
 (d) INTEGER RANGE -2 TO 15;

Note that ****** stand for the power-of symbol.

1.16: Determine the error lines (Y/N) in the VHDL code below and explain what is wrong, or give correct code.

VHDL code	Error (Y/N)	Give reason
LIBRARY ieee; /* Using predefined packages */		
ENTITY error IS		
PORTS (x: in BIT; c: in BIT;		
Z1: out INTEGER; z2 : out BIT);		
END error		
ARCHITECTURE error OF has IS		
SIGNAL s ; w : BIT;		
BEGIN		
w := c;		
Z1 <= x;		
P1: PROCESS (x)		
BEGIN		
IF c='1' THEN		
x <= z2;		
END PROCESS P0;		
END OF has;		

1.17: Determine the error lines (Y/N) in the VHDL code below, and explain what is wrong, or give correct code.

VHDL code	Error (Y/N)	Give reason
LIBRARY ieee; /* Using predefined packages */		
USE altera.std_logic_1164.ALL;		
ENTITY srhifreg IS		
GENERIC (WIDTH : POSITIVE = 4);		
PORT(clk, din : IN STD_LOGIC;		
dout : OUT STD_LOGIC);		
END;		
ARCHITECTURE a OF shiftrig IS		
COMPONENT d_ff		
PORT (clock, d : IN std_logic;		
q : OUT std_logic);		
END d_ff;		
SIGNAL b : logic_vector(0 TO width-1);		
BEGIN		
d1: d_ff PORT MAP (clk, b(0), din);		
g1: FOR j IN 1 TO width-1 GENERATE		
d2: d-ff		
PORT MAP(clk => clock,		
din => b(j-1),		
q => b(j));		
END GENERATE d2;		
dout <= b(width);		
END a;		

1.18: Determine for the following process statements

- (a) the synthesized circuit and label I/O ports
- (b) the cost of the design assuming a cost 1 per adder/subtractor
- (c) the critical (i.e., worst-case) path of the circuit for each process. Assume a delay of 1 for an adder or subtractor.

```
-- QUIZ VHDL2graph for DSP with FPGAs
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY qv2g IS
  PORT(a, b, c, d : IN std_logic_vector(3 DOWNTO 0);
       u, v, w, x, y, z : OUT std_logic_vector(3 DOWNTO 0));
END;
ARCHITECTURE a OF qv2g IS BEGIN

  P0: PROCESS(a, b, c, d)
  BEGIN
    u <= a + b - c + d;
  END PROCESS;

  P1: PROCESS(a, b, c, d)
  BEGIN
    v <= (a + b) - (c - d);
  END PROCESS;

  P2: PROCESS(a, b, c)
  BEGIN
    w <= a + b + c;
    x <= a - b - c;
  END PROCESS;

  P3: PROCESS(a, b, c)
  VARIABLE t1 : std_logic_vector(3 DOWNTO 0);
  BEGIN
    t1 := b + c;
    y <= a + t1;
    z <= a - t1;
  END PROCESS;
END;
```

- 1.19:** (a) Develop a functions for zero- and sign extension called `ZERO_EXT(ARG,SIZE)` and `SIGN_EXT(ARG,SIZE)` for the `STD_LOGIC_VECTOR` data type.
 (b) Develop “*” and “/” function overloading to implement multiply and divide operation for the `STD_LOGIC_VECTOR` data type.
 (c) Use the test bench shown in Fig. 1.28 to verify the correct functionality.

- 1.20:** (a) Design a function library for the `STD_LOGIC_VECTOR` data type that implement the following operation (defined in VHDL-1993 only for the `BIT_VECTOR` data type):
 (a) SRL (b) SRA (c) SLL (d) SLA
 (e) Use the test bench shown in Fig. 1.29 to verify the correct functionality. Note the high impedance values Z that are part of the `STD_LOGIC_VECTOR` data type but are not included in the `BIT_VECTOR` data type. A left/right shift by a negative value should be replaced by the appropriate right/left shift of the positive amount inside your function.

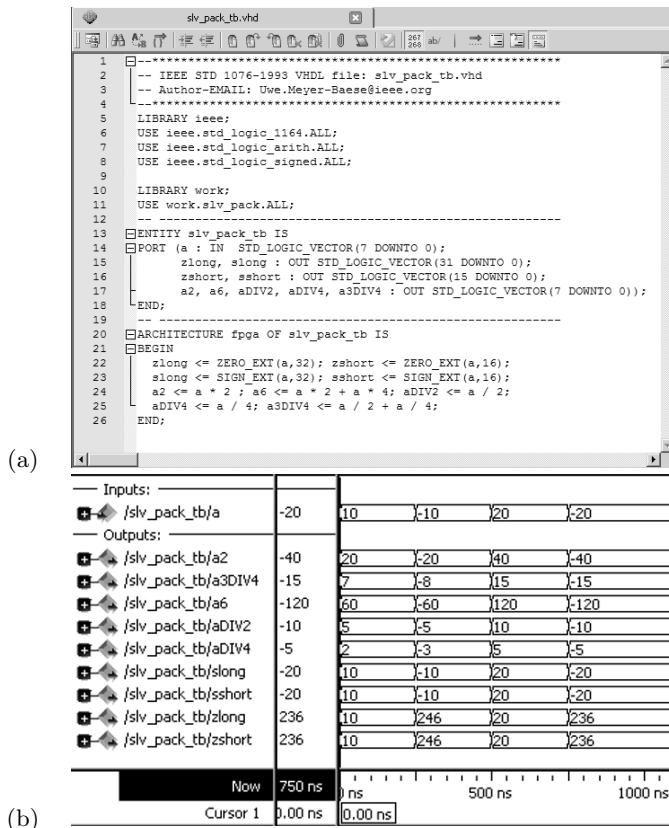


Fig. 1.28. STD_LOGIC_VECTOR package test bench. (a) HDL code. (b) Functional simulation result

1.21: Determine for the following PROCESS statements the synthesized circuit type (combinational, latch, D-flip-flop, or T-flip-flop) and the function of **a**, **b**, and **c**, i.e., clock, a-synchronous set (AS) or reset (AR) or synchronous set (SS) or reset (SR). Use the table below to specify the classification.

```

LIBRARY ieee; USE ieee.std_logic_1164.ALL;

ENTITY quiz IS
    PORT(a, b, c : IN std_logic;
         d : IN std_logic_vector(0 TO 5);
         q : BUFFER std_logic_vector(0 TO 5));
END quiz;
ARCHITECTURE a OF quiz IS BEGIN
    PO: PROCESS (a)
    BEGIN
        IF rising_edge(a) THEN
            q(0) <= d(0);
        END IF;
    
```

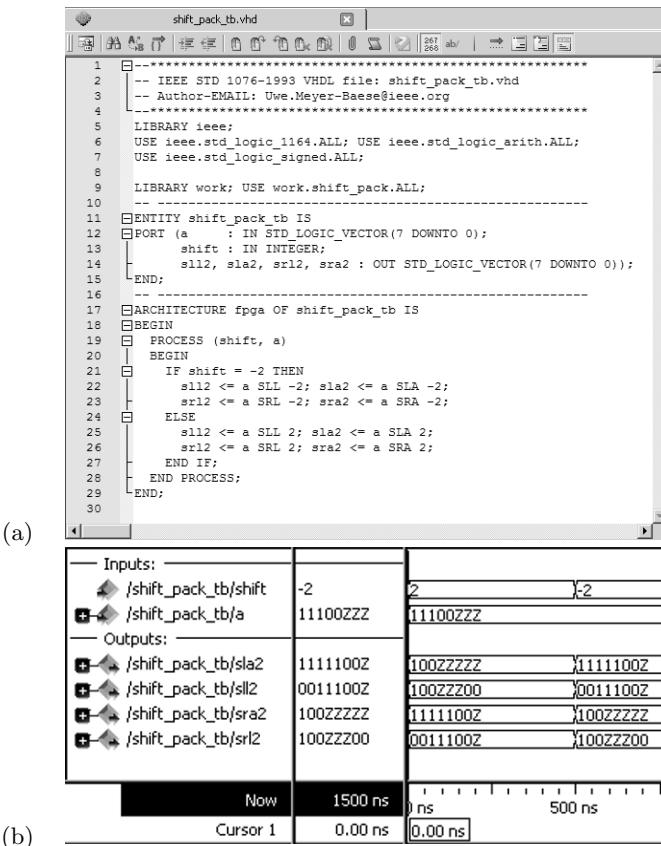


Fig. 1.29. STD_LOGIC_VECTOR shift library test bench. (a) HDL code. (b) Functional simulation result

```

END PROCESS P0;

P1: PROCESS (a, d)
BEGIN
  IF a= '1' THEN q(1) <= d(1);
  ELSE q(1) <= '1';
  END IF;
END PROCESS P1;

P2: PROCESS (a, b, c, d)
BEGIN
  IF a = '1' THEN q(2) <= '0';
  ELSE IF rising_edge(b) THEN
    IF c = '1' THEN q(2) <= '1';
    ELSE q(2) <= d(1);
    END IF;
  END IF;
END IF;

```

```

        END IF;
END PROCESS P2;

P3: PROCESS (a, b, d)
BEGIN
    IF a = '1' THEN q(3) <= '1';
    ELSE IF rising_edge(b) THEN
        IF c = '1' THEN q(3) <= '0';
        ELSE q(3) <= not q(3);
        END IF;
        END IF;
    END IF;
END PROCESS P3;

P4: PROCESS (a, d)
BEGIN
    IF a = '1' THEN q(4) <= d(4);
    END IF;
END PROCESS P4;

P5: PROCESS (a, b, d)
BEGIN
    IF rising_edge(a) THEN
        IF b = '1' THEN q(5) <= '0';
        ELSE q(5) <= d(5);
        END IF;
    END IF;
END PROCESS P5;

```

Process	Circuit type	CLK	AS	AR	SS	SR
P0						
P1						
P2						
P3						
P4						
P5						

1.22: Given the following MATLAB instructions,

```

a=-1:2:5
b=[ones(1,2),zeros(1,2)]
c=a*a'
d=a.*a
e=a'*a
f=conv(a,b)
g=fft(b)
h=ifft(fft(a).*fft(b))

```

determine a-h.

2. Computer Arithmetic

2.1 Introduction

In computer arithmetic two fundamental design principles are of great importance: number representation and the implementation of algebraic operations [29–33]. We will first discuss possible number representations, (e.g., fixed-point or floating-point), then basic operations like adder and multiplier, and finally efficient implementation of more difficult operations such as square roots, and the computation of trigonometric functions using the CORDIC algorithm or MAC calls.

FPGAs allow a wide variety of computer arithmetic implementations for the desired digital signal processing algorithms, because of the physical bit-level programming architecture. This contrasts with the programmable digital signal processors (PDSPs), with the fixed multiply accumulator core. Careful choice of the bit width in FPGA design can result in substantial savings.

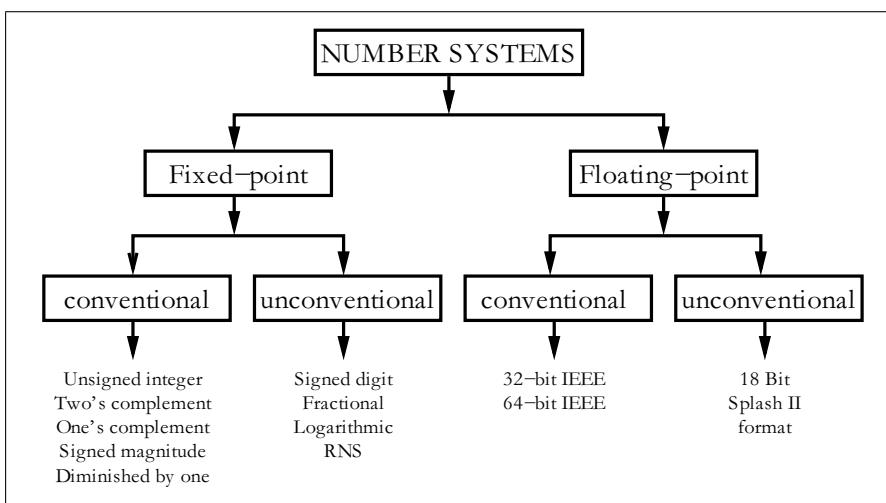


Fig. 2.1. Survey of number representations

2.2 Number Representation

Deciding whether fixed- or floating-point is more appropriate for the problem must be done carefully, preferably at an early phase in the project. In general, it can be assumed that fixed-point implementations have higher speed and lower cost, while floating-point has higher dynamic range and no need for scaling, which may be attractive for more complicated algorithms. Figure 2.1 is a survey of conventional and less conventional fixed- and floating-point number representations. Both systems are covered by a number of standards but may, if desired, be implemented in a proprietary form.

2.2.1 Fixed-Point Numbers

We will first review the fixed-point number systems shown in Fig. 2.1. Table 2.1 shows the 3-bit coding for the 5 different integer representations.

Unsigned Integer

Let X be an N -bit unsigned binary number. Then the range is $[0, 2^N - 1]$ and the representation is given by:

$$X = \sum_{n=0}^{N-1} x_n 2^n, \quad (2.1)$$

where x_n is the n^{th} binary digit of X (i.e., $x_n \in [0, 1]$). The digit x_0 is called the least significant bit (LSB) and has a relative weight of unity. The digit x_{N-1} is the most significant bit (MSB) and has a relative weight of 2^{N-1} .

Signed-Magnitude (SM)

In signed-magnitude systems the magnitude and the sign are represented separately. The first bit x_{N-1} (i.e., the MSB) represents the sign and the remaining $N - 1$ bits the magnitude. The representation becomes:

$$X = \begin{cases} \sum_{n=0}^{N-2} x_n 2^n & X \geq 0 \\ -\sum_{n=0}^{N-2} x_n 2^n & X < 0. \end{cases} \quad (2.2)$$

The range of this representation is $[-(2^{N-1} - 1), 2^{N-1} - 1]$. The advantage of the signed-magnitude representation is simplified prevention of overflows, but the disadvantage is that addition must be split depending on which operand is larger.

Table 2.1. Conventional coding of signed binary numbers

Binary	2C	1C	D1	SM	Bias
011	3	3	4	3	0
010	2	2	3	2	-1
001	1	1	2	1	-2
000	0	0	1	0	-3
111	-1	-0	-1	-3	4
110	-2	-1	-2	-2	3
101	-3	-2	-3	-1	2
100	-4	-3	-4	-0	1
1000	-	-	0	-	-

Two's Complement (2C)

An N -bit two's complement representation of a signed integer, over the range $[-2^{N-1}, 2^{N-1} - 1]$, is given by:

$$X = \begin{cases} \sum_{n=0}^{N-2} x_n 2^n & X \geq 0 \\ -2^{N-1} + \sum_{n=0}^{N-2} x_n 2^n & X < 0. \end{cases} \quad (2.3)$$

The two's complement (2C) system is by far the most popular signed numbering system in DSP use today. This is because it is possible to add several signed numbers, and as long as the final sum is in the N -bit range, we can ignore *any* overflow in the arithmetic. For instance, if we add two 3-bit numbers as follows:

$$\begin{array}{rcl} 3_{10} & \longleftrightarrow & 0\ 1\ 1_{2C} \\ -2_{10} & \longleftrightarrow & 1\ 1\ 0_{2C} \\ 1_{10} & \longleftrightarrow & 1.\ 0\ 0\ 1_{2C} \end{array}$$

the overflow can be ignored. All computations are modulo 2^N . It follows that it is possible to have intermediate values that cannot be correctly represented, but if the final value is valid then the result is correct. For instance, if we add the 3-bit numbers $2 + 2 - 3$, we would have an intermediate value of $010 + 010 = 100_{2C}$, i.e., -4_{10} , but the result $100 - 011 = 100 + 101 = 001_{2C}$ is correct.

Two's complement numbers can also be used to implement modulo 2^N arithmetic without any change in the arithmetic. This is what we will use in Chap. 5 to design CIC filters.

One's Complement (1C)

An N -bit one's complement system (1C) can represent integers over the range $[-(2^{N-1} + 1), 2^{N-1} - 1]$. In a one's complement code, positive and negative

numbers have bit-by-bit complement representations including for the sign bit. There is, in fact a redundant representation of zero (see Table 2.1). The representation of signed numbers in a 1C system is formally given by:

$$X = \begin{cases} \sum_{n=0}^{N-2} x_n 2^n & X \geq 0 \\ -2^{N-1} + 1 + \sum_{n=0}^{N-2} x_n 2^n & X < 0. \end{cases} \quad (2.4)$$

For example, the three-bit 1C representation of the numbers -3 to 3 is shown in the third column of Table 2.1.

From the following simple example

$$\begin{array}{rcl} 3_{10} & \longleftrightarrow & 0\ 1\ 1_{1C} \\ -2_{10} & \longleftrightarrow & 1\ 0\ 1_{1C} \\ 1_{10} & \longleftrightarrow & 1.\ 0\ 0_{1C} \\ \text{Carry} & \hookrightarrow \rightarrow \rightarrow & 1_{1C} \\ 1_{10} & \longleftrightarrow & 0\ 0\ 1_{1C} \end{array}$$

we remember that in one's complement a “carry wrap-around” addition is needed. A carry occurring at the MSB must be added to the LSB to get the correct final result.

The system can, however, efficiently be used to implement modulo $2^N - 1$ arithmetic without correction. As a result, one's complement has specialized value in implementing selected DSP algorithms (e.g., Mersenne transforms over the integer ring $2^N - 1$ [34]).

Diminished One System (D1)

A diminished one (D1) system is a biased system. The positive numbers are, compared with the 2C, diminished by 1. The range for $(N+1)$ -bit D1 numbers is $[-2^{N-1}, 2^{N-1}]$, excluding 0. The coding rule for a D1 system is defined as follows:

$$X = \begin{cases} \sum_{n=0}^{N-2} x_n 2^n + 1 & X > 0 \\ -2^{N-1} + \sum_{n=0}^{N-2} x_n 2^n & X < 0 \\ 2^N & X = 0. \end{cases} \quad (2.5)$$

From adding two D1 numbers

$$\begin{array}{rcl} 3_{10} & \longleftrightarrow & 0\ 1\ 0_{D1} \\ -2_{10} & \longleftrightarrow & 1\ 1\ 0_{D1} \\ 1_{10} & \longleftrightarrow & 1.\ 0\ 0_{D1} \\ \text{Carry} & \hookrightarrow \boxed{\text{NOT}} \rightarrow 0_{D1} \\ 1_{10} & \longleftrightarrow & 0\ 0\ 0_{D1} \end{array}$$

we see that, for D1 a complement and add of the *inverted* carry must be computed.

D1 numbers can efficiently be used to implement modulo $2^N + 1$ arithmetic without any change in the arithmetic. This fact can be used to implement Fermat NTTs in the ring $2^N + 1$ [34].

Bias System

The biased number system has a bias for all numbers. The bias value is usually in the middle of the binary range, i.e., bias = $2^{N-1} - 1$. For a 3-bit system, for instance the bias would be $2^{3-1} - 1 = 3$. The range for N -bit biased numbers is $[-2^{N-1} - 1, 2^{N-1}]$. Zero is coded as the bias. The coding rule for a biased system is defined as follows:

$$X = \sum_{n=0}^{N-1} x_n 2^n - \text{bias}. \quad (2.6)$$

From adding two biased numbers

$$\begin{array}{rcl} 3_{10} & \longleftrightarrow & 1\ 1\ 0_{\text{bias}} \\ +(-2_{10}) & \longleftrightarrow & 0\ 0\ 1_{\text{bias}} \\ 4_{10} & \longleftrightarrow & 1\ 1\ 1_{\text{bias}} \\ -\text{bias} & \longleftrightarrow & 0\ 1\ 1_{\text{bias}} \\ 1_{10} & \longleftrightarrow & 1\ 0\ 0_{\text{bias}} \end{array}$$

we see that, for each addition the bias needs to be subtracted, while for every subtraction the bias needs to be added.

Bias numbers can efficiently be used to simplify comparison of numbers. This fact will be used in Sect. 2.2.3 (p. 75) for coding the exponent of floating-point numbers.

2.2.2 Unconventional Fixed-Point Numbers

In the following we continue the review of number systems according to Fig. 2.1 (p. 57). The unconventional fixed-point number systems discussed in the following are not as often used as for instance the 2C system, but can yield significant improvements for particular applications or problems.

Signed Digit Numbers (SD)

The signed digit (SD) system differs from the traditional binary systems presented in the previous section in the fact that it is ternary valued (i.e., digits have the value $\{0, 1, -1\}$, where -1 is sometimes denoted as $\bar{1}$).

SD numbers have proven to be useful in carry-free adders or multipliers with less complexity, because the effort in multiplication can typically be estimated through the number of nonzero elements, which can be reduced by using SD numbers. Statistically, half the digits in the two's complement

coding of a number are zero. For an SD code, the density of zeros increases to two thirds as the following example shows:

Example 2.1: SD Coding

Consider coding the decimal number $15 = 1111_2$ using a 5-bit binary and an SD code. Their representations are as follows:

- 1) $15_{10} = 16_{10} - 1_{10} = 1000\bar{1}_{SD}$
- 2) $15_{10} = 16_{10} - 2_{10} + 1_{10} = 100\bar{1}1_{SD}$
- 3) $15_{10} = 16_{10} - 4_{10} + 3_{10} = 10\bar{1}11_{SD}$
- 4) etc.

2.1

The SD representation, unlike a 2C code, is nonunique. We call a *canonic signed digit* system (CSD) the system with the minimum number of non-zero elements. The following algorithm can be used to produce a classical CSD code.

Algorithm 2.2: Classical CSD Coding

Starting with the LSB substitute all 1 sequences equal or larger than two, with $10\dots0\bar{1}$.

This CSD coding is the basis for the C utility program `csd.exe`¹ on the CD-ROM. This classical CSD code is also unique and an additional property is that the resulting representation has at least one zero between two digits, which may have values 1, $\bar{1}$, or 0.

Example 2.3: Classical CSD Code

Consider again coding the decimal number 15 using a 5-bit binary and a CSD code. Their representations are: $1111_2 = 1000\bar{1}_{CSD}$. We notice from a comparison with the SD coding from Example 2.1 that only the first representation is a CSD code.

As another example consider the coding of

$$27_{10} = 11011_2 = 1110\bar{1}_{SD} = 100\bar{1}0\bar{1}_{CSD}. \quad (2.7)$$

We note that, although the first substitution of $011 \rightarrow 10\bar{1}$ does not reduce the complexity, it produces a length-three strike, and the complexity reduces from three additions to two subtractions.

2.3

On the other hand, the classical CSD coding does not always produce the optimal CSD coding in terms of hardware complexity, because in Algorithm 2.2 additions are also substituted by subtractions, when there should be no such substitution. For instance 011_2 is coded as $10\bar{1}_{CSD}$, and if this coding is used to produce a constant multiplier the subtraction will need a full-adder instead of a half-adder for the LSB. The CSD coding given in the following

¹ You need to copy the program to your harddrive first because the program writes out the results in a file `csd.dat`; you cannot start it from the CD directly.

will produce a CSD coding with the minimum number of nonzero terms, but also with the minimum number of subtractions.

Algorithm 2.4: Optimal CSD Coding

- 1) Starting with the LSB substitute all 1 sequences larger than two with $10\dots0\bar{1}$. Also substitute 1011 with $110\bar{1}$.
- 2) Starting with the MSB, substitute $10\bar{1}$ with 011.

Fractional (CSD) Coding

Many DSP algorithms require the implementation of fractional numbers. Think for instance of trigonometric coefficient like sine or cosine coefficients. Implementation via integer numbers only would result in a large quantization error. The question then is, can we also use the CSD coding to reduce the implementation effort of a fractional constant coefficient? The answer is yes, but we need to be a little careful with the ordering of the operands. In VHDL the analysis of an expression is usually done from left to right, which means an expression like $y = 7 \times x/8$ is implemented as $y = (7 \times x)/8$, and equivalently the expression $y = x/8 \times 7$ is implemented as $y = (x/8) \times 7$. The latter term unfortunately will produce a large quantization error, since the evaluation of $x/8$ is in fact synthesized by the tool² as a right shift by three bits, so we will lose the lower three bits of our input x in the computation that follows. Let us demonstrate this with a small HDL design example.

Example 2.5: Fractional CSD Coding

Consider coding the fractional decimal number $0.875 = 7/8$ using a fractional 4-bit binary and CSD code. The 7 can be implemented more efficiently in CSD as $7 = 8 - 1$ and we want to determine the quantization error of the following four mathematically equivalent representations, which give different synthesis results:

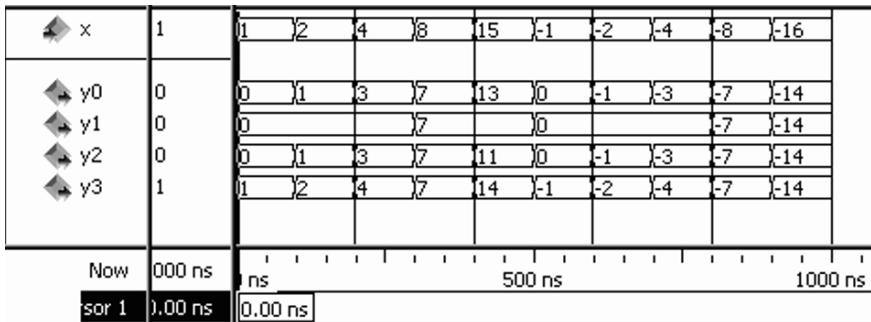
$$\begin{aligned}y_0 &= 7 \times x/8 = (7 \times x)/8 \\y_1 &= x/8 \times 7 = (x/8) \times 7 \\y_2 &= x/2 + x/4 + x/8 = ((x/2) + (x/4)) + (x/8) \\y_3 &= x - x/8 = x - (x/8)\end{aligned}$$

Using parenthesis in the above equations it is shown how the HDL tool will group the expressions. Multiply and divide have a higher priority than add and subtract and the evaluation is from left to right. The VHDL code³ of the constant coefficient fractional multiplier is shown next.

```
ENTITY cmul7p8 IS
    PORT(x : IN INTEGER RANGE -16 TO 15; -- System input
         y0, y1, y2, y3 : OUT INTEGER RANGE -16 TO 15);
```

² Most HDL tools only support dividing by power-of-two values, which can be designed using a shifter, see Sect. 2.5, p. 93.

³ The equivalent Verilog code `cmul7p8.v` for this example can be found in Appendix A on page 797. Synthesis results are shown in Appendix B on page 881.

**Fig. 2.2.** Simulation results for fractional CSD coding

```

END;                                     -- The 4 system outputs y=7*x/8
-----
ARCHITECTURE fpga OF cmul7p8 IS
BEGIN

    y0 <= 7 * x / 8;
    y1 <= x / 8 * 7;
    y2 <= x/2 + x/4 + x/8;
    y3 <= x - x/8;

END fpga;

```

The design uses 48 LEs and no embedded multiplier. A registered performance cannot be measured since there is no register-to-register path. The simulated results of the fractional constant coefficient multiplier is shown in Fig. 2.2. Note the large quantization error for y_1 . Looking at the results for the input value $x = 4$, we can also see that the CSD coding y_3 shows rounding to the next largest integer, while y_0 and y_2 show rounding to the next smallest integer. For negative value (e.g., -4) we see that the CSD coding y_3 shows rounding to the next smallest (i.e., -4) integer, while y_0 and y_2 show rounding to the next largest (i.e., -3) integer.

2.5

Carry-Free Adder

The SD number representation can also be used to implement a carry-free adder. Tagaki et al. [35] introduced the scheme presented in Table 2.2. Here, u_k is the interim sum and c_k is the carry of the k^{th} bit (i.e., to be added to u_{k+1}).

Example 2.6: Carry-Free Addition

The addition of 29 to -9 in the SD system is performed below.

Table 2.2. Adding carry-free binaries using the SD representation

$x_k y_k$	00	01	01	0̄1	0̄1	11	1̄1
$x_{k-1} y_{k-1}$	—	neither is $\bar{1}$	at least one is $\bar{1}$	neither is $\bar{1}$	at least one is $\bar{1}$	—	—
c_k	0	1	0	0	$\bar{1}$	1	$\bar{1}$
u_k	0	$\bar{1}$	1	$\bar{1}$	1	0	0

$$\begin{array}{r}
 1\ 0\ 0\ \bar{1}\ 0\ 1\ x_k \\
 +\ 0\ \bar{1}\ 1\ \bar{1}\ 1\ 1\ y_k \\
 \hline
 0\ 0\ 0\ \bar{1}\ 1\ 1\ c_k \\
 1\ \bar{1}\ 1\ 0\ \bar{1}\ 0\ u_k \\
 \hline
 1\ \bar{1}\ 0\ 1\ 0\ 0\ s_k
 \end{array}$$

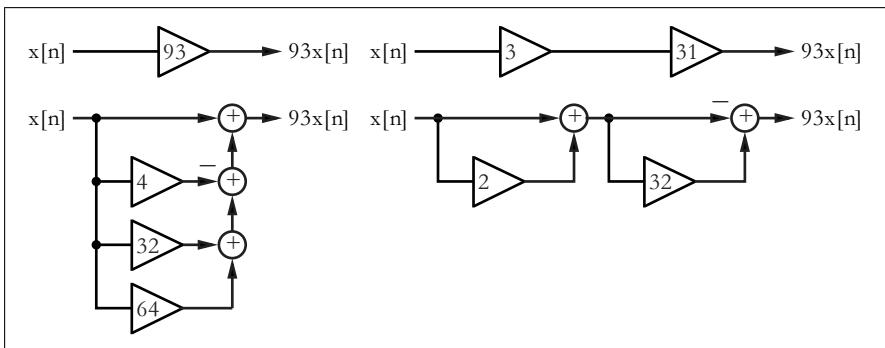
2.6

However, due to the ternary logic burden, implementing Table 2.2 with FPGAs requires four-input operands for the c_k and u_k . This translates into a $2^8 \times 4$ -bit LUT when implementing Table 2.2.

Multiplier Adder Graph (MAG)

We have seen that the cost of multiplication is a direct function of the number of nonzero elements a_k in A . The CSD system minimizes this cost. The CSD is also the basis for the Booth multiplier [29] discussed in Exercise 2.2 (p. 169).

It can, however, sometimes be more efficient first to factor the coefficient into several factors, and realize the individual factors in an optimal CSD sense [36–39]. Figure 2.3 illustrates this option for the coefficient 93. The direct binary and CSD codes are given by $93_{10} = 1011101_2 = 1100\bar{1}01_{\text{CSD}}$,

**Fig. 2.3.** Two realizations for the constant factor 93

with the 2C requiring four adders, and the CSD requiring three adders. The coefficient 93 can also be represented as $93 = 3 \times 31$, which requires one adder for each factor (see Fig. 2.3). The complexity for the factor number is reduced to two. There are several ways to combine these different factors. The number of adders required is often referred to as the cost of the constant coefficient multiplier. Figure 2.4, suggested by Dempster et al. [38], shows all possible configurations for one to four adders. Using this graph, all coefficients with a cost ranging from one to four can be synthesized with $k_i \in \mathbb{N}_0$, according to:

$$\begin{aligned} \textbf{Cost 1: } & 1) \quad A = 2^{k_0}(2^{k_1} \pm 2^{k_2}) \\ \textbf{Cost 2: } & 1) \quad A = 2^{k_0}(2^{k_1} \pm 2^{k_2} \pm 2^{k_3}) \\ & 2) \quad A = 2^{k_0}(2^{k_1} \pm 2^{k_2})(2^{k_3} \pm 2^{k_4}) \\ \textbf{Cost 3: } & 1) \quad A = 2^{k_0}(2^{k_1} \pm 2^{k_2} \pm 2^{k_3} \pm 2^{k_4}) \\ & \vdots \end{aligned}$$

Using this technique, Table 2.3 shows the optimal coding for all 8-bit integers having a cost between zero and three [5].

Logarithmic Number System (LNS)

The logarithmic number system (LNS) [40, 41] is analogous to the floating-point system with a fixed mantissa and a fractional exponent. In the LNS, a number x is represented as:

$$X = \pm r^{\pm e_x}, \quad (2.8)$$

where r is the system's radix, and e_x is the LNS exponent. The LNS format consists of a sign-bit for the number and exponent, and an exponent assigned I integer bits and F fractional bits of precision. The format in graphical form is shown below:

Sign	Exponent	Exponent integer bits I	Exponent fractional bits F
S_X	sign S_e		

The LNS, like floating-point, carries a nonuniform precision. Small values of x are highly resolved, while large values of x are more coarsely resolved as the following example shows.

Example 2.7: LNS Coding

Consider a radix-2 9-bit LNS word with two sign-bits, three bits for integer precision and four-bit fractional precision. How can, for instance, the LNS coding 00 011.0010 be translated into the real number system? The two sign bits indicate that the whole number and the exponent are positive. The integer part is 3 and the fractional part $2^{-3} = 1/8$. The real number representation is therefore $2^{3+1/8} = 2^{3.125} = 8.724$. We find also that $-2^{3.125} = 10\ 011.0010$ and $2^{-3.125} = 01\ 100.1110$. Note that the exponent is represented in fractional two's complement format. The largest number that can be represented with this 9-bit LNS format is $2^{8-1/16} \approx 2^8 = 256$ and

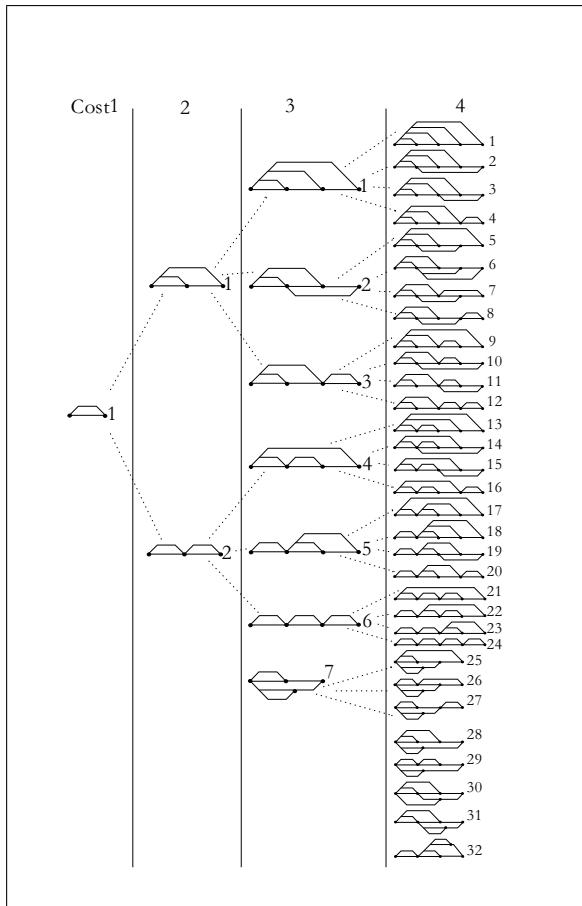


Fig. 2.4. Possible cost one to four graphs. Each node is either an adder or subtractor and each edge is associated with a power-of-two factor. (© IEEE [38])

the smallest is $2^{-8} = 0.0039$, as graphically interpreted in Fig. 2.5a. In contrast, an 8-bit plus sign fixed-point number has a maximal positive value of $2^8 - 1 = 255$, and the smallest nonzero positive value is one. A comparison of the two 9-bit systems is shown in Fig. 2.5b.

2.7

The historical attraction of the LNS lies in its ability to efficiently implement multiplication, division, square-rooting, or squaring. For example, the product $C = A \times B$, where A , B , and C are LNS words, is given by:

$$C = r^{e_a} \times r^{e_b} = r^{e_a + e_b} = r^{e_c}. \quad (2.9)$$

That is, the exponent of the LNS product is simply the sum of the two exponents. Division and high-order operations immediately follow. Unfortunately,

Table 2.3. Cost C (i.e., number of adders) for all 8-bit numbers using the multiplier adder graph (MAG) technique

C	Coefficient	
0	1, 2, 4, 8, 16, 32, 64, 128, 256	
1	3, 5, 6, 7, 9, 10, 12, 14, 15, 17, 18, 20, 24, 28, 30, 31, 33, 34, 36, 40, 48, 56, 60, 62, 63, 65, 66, 68, 72, 80, 96, 112, 120, 124, 126, 127, 129, 130, 132, 136, 144, 160, 192, 224, 240, 248, 252, 254, 255	
2	11, 13, 19, 21, 22, 23, 25, 26, 27, 29, 35, 37, 38, 39, 41, 42, 44, 46, 47, 49, 50, 52, 54, 55, 57, 58, 59, 61, 67, 69, 70, 71, 73, 74, 76, 78, 79, 81, 82, 84, 88, 92, 94, 95, 97, 98, 100, 104, 108, 110, 111, 113, 114, 116, 118, 119, 121, 122, 123, 125, 131, 133, 134, 135, 137, 138, 140, 142, 143, 145, 146, 148, 152, 156, 158, 159, 161, 162, 164, 168, 176, 184, 188, 190, 191, 193, 194, 196, 200, 208, 216, 220, 222, 223, 225, 226, 228, 232, 236, 238, 239, 241, 242, 244, 246, 247, 249, 250, 251, 253	
3	43, 45, 51, 53, 75, 77, 83, 85, 86, 87, 89, 90, 91, 93, 99, 101, 102, 103, 105, 106, 107, 109, 115, 117, 139, 141, 147, 149, 150, 151, 153, 154, 155, 157, 163, 165, 166, 167, 169, 170, 172, 174, 175, 177, 178, 180, 182, 183, 185, 186, 187, 189, 195, 197, 198, 199, 201, 202, 204, 206, 207, 209, 210, 212, 214, 215, 217, 218, 219, 221, 227, 229, 230, 231, 233, 234, 235, 237, 243, 245	
4	171, 173, 179, 181, 203, 205, 211, 213	
Minimum costs through factorization		
$45 = 5 \times 9, 51 = 3 \times 17, 75 = 5 \times 15, 85 = 5 \times 17, 90 = 2 \times 9 \times 5, 93 = 3 \times 31, 99 = 3 \times 33, 102 = 2 \times 3 \times 17, 105 = 7 \times 15, 150 = 2 \times 5 \times 15, 153 = 3 \times 51, 155 = 5 \times 31, 165 = 5 \times 33, 170 = 2 \times 5 \times 17, 180 = 4 \times 5 \times 9, 186 = 2 \times 3 \times 31, 189 = 7 \times 9, 195 = 3 \times 65, 198 = 2 \times 3 \times 33, 204 = 4 \times 3 \times 17, 210 = 2 \times 7 \times 15, 217 = 7 \times 31, 231 = 7 \times 33$		
2	$171 = 3 \times 57, 173 = 8 + 165, 179 = 51 + 128, 181 = 1 + 180, 211 = 1 + 210, 213 = 3 \times 71, 205 = 5 \times 41, 203 = 7 \times 29$	

addition or subtraction are by comparison far more complex. Addition and subtraction operations are based on the following procedure, where it is assumed that $A > B$.

$$C = A + B = 2^{e_a} + 2^{e_b} = 2^{e_a} \underbrace{\left(1 + 2^{e_b - e_a}\right)}_{\Phi^+(\Delta)} = 2^{e_c}. \quad (2.10)$$

Solving for the exponent e_c , one obtains $e_c = e_a + \phi^+(\Delta)$ where $\Delta = e_b - e_a$ and $\phi^+(u) = \log_2(\Phi^+(\Delta))$. For subtraction a similar table, $\phi^-(u) = \log_2(\Phi^-(\Delta))$, $\Phi^-(\Delta) = (1 - 2^{e_b - e_a})$, can be used. Such tables have been historically used for rational numbers as described in “Logarithmorum Completus,” Jurij Vega (1754–1802), containing tables computed by Zech. As a result, the term $\log_2(1 - 2^u)$ is usually referred to as a Zech logarithm.

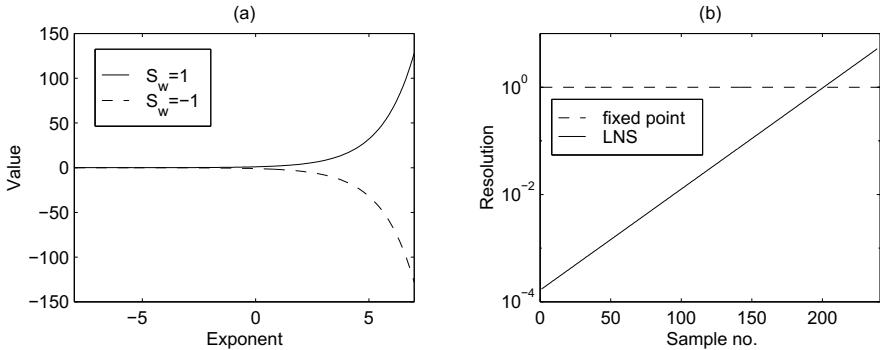


Fig. 2.5. LNS processing. (a) Values. (b) Resolution

LNS arithmetic is performed in the following manner [40]. Let $A = 2^{e_a}, B = 2^{e_b}, C = 2^{e_c}$, with S_A, S_B, S_C denoting the sign-bit for each word:

Operation	Action	
Multiply	$C = A \times B$	$e_c = e_a + e_b; S_C = S_A \text{ XOR } S_B$
Divide	$C = A/B$	$e_c = e_a - e_b; S_C = S_A \text{ XOR } S_B$
Add	$C = A + B$	$e_c = \begin{cases} e_a + \phi^+(e_b - e_a) & A \geq B \\ e_b + \phi^+(e_a - e_b) & B > A \end{cases}$
Subtract	$C = A - B$	$e_c = \begin{cases} e_a + \phi^-(e_b - e_a) & A \geq B \\ e_b + \phi^-(e_a - e_b) & B > A \end{cases}$
Square root	$C = \sqrt{A}$	$e_c = e_a/2$
Square	$C = A^2$	$e_c = 2e_a$

Methods have been developed to reduce the necessary table size for the Zech logarithm by using partial tables [40] or using linear interpolation techniques [42]. These techniques are beyond the scope of the discussion presented here.

Residue Number System (RNS)

The RNS is actually an ancient algebraic system whose history can be traced back 2000 years. The RNS is an integer arithmetic system in which the primitive operations of addition, subtraction, and multiplication are defined. The primitive operations are performed concurrently within noncommunicating small-wordlength channels [43, 44]. An RNS system is defined with respect to a positive integer basis set $\{m_1, m_2, \dots, m_L\}$, where the m_l are all relatively (pairwise) prime. The dynamic range of the resulting system is M , where $M = \prod_{l=1}^L m_l$. For signed-number applications, the integer value of X is assumed to be constrained to $X \in [-M/2, M/2)$. RNS arithmetic is defined within a ring isomorphism:

$$\mathbb{Z}_M \cong \mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_L}, \quad (2.11)$$

where $\mathbb{Z}_M = \mathbb{Z}/(M)$ corresponds to the ring of integers modulo M , called the residue class mod M . The mapping of an integer X into an RNS L -tuple $X \leftrightarrow (x_1, x_2, \dots, x_L)$ is defined by $x_l = X \bmod m_l$, for $l = 1, 2, \dots, L$. Defining \square to be the algebraic operations $+$, $-$ or $*$, it follows that, if $Z, X, Y \in \mathbb{Z}_M$, then:

$$Z = X \square Y \bmod M \quad (2.12)$$

is isomorphic to $Z \leftrightarrow (z_1, z_2, \dots, z_L)$. Specifically:

$$\begin{array}{ccc} X & \xrightarrow{(m_1, m_2, \dots, m_L)} & (\langle X \rangle_{m_1}, \langle X \rangle_{m_2}, \dots, \langle X \rangle_{m_L}) \\ Y & \xleftarrow{(m_1, m_2, \dots, m_L)} & (\langle Y \rangle_{m_1}, \langle Y \rangle_{m_2}, \dots, \langle Y \rangle_{m_L}) \\ \hline Z = X \square Y & \xrightarrow{(m_1, m_2, \dots, m_L)} & (\langle X \square Y \rangle_{m_1}, \langle X \square Y \rangle_{m_2}, \dots, \langle X \square Y \rangle_{m_L}). \end{array}$$

As a result, RNS arithmetic is pairwise defined. The L elements of $Z = (X \square Y) \bmod M$ are computed concurrently within L small-wordlength mod (m_l) channels whose width is bounded by $w_l = \lceil \log_2(m_l) \rceil$ bits (typical 4 to 8 bits). In practice, most RNS arithmetic systems use small RAM or ROM tables to implement the modular mappings $z_l = x_l \square y_l \bmod m_l$.

Example 2.8: RNS Arithmetic

Consider an RNS system based on the relatively prime moduli set $\{2, 3, 5\}$ having a dynamic range of $M = 2 \times 3 \times 5 = 30$. Two integers in \mathbb{Z}_{30} , say 7_{10} and 4_{10} , have RNS representations $7 = (1, 1, 2)_{\text{RNS}}$ and $4 = (0, 1, 4)_{\text{RNS}}$, respectively. Their sum, difference, and products are 11, 3, and 28, respectively, which are all within \mathbb{Z}_{30} . Their computation is shown below.

$$\begin{array}{c} 7 \xleftarrow{(2,3,5)} (1, 1, 2) \quad 7 \xleftarrow{(2,3,5)} (1, 1, 2) \quad 7 \xleftarrow{(2,3,5)} (1, 1, 2) \\ +4 \xleftarrow{(2,3,5)} +(0, 1, 4) \quad -4 \xleftarrow{(2,3,5)} -(0, 1, 4) \quad \times 4 \xleftarrow{(2,3,5)} \times(0, 1, 4) \\ \hline 11 \xleftarrow{(2,3,5)} (1, 2, 1) \quad 3 \xleftarrow{(2,3,5)} (1, 0, 3) \quad 28 \xleftarrow{(2,3,5)} (0, 1, 3). \end{array}$$

2.8

RNS systems have been built as custom VLSI devices [45], GaAs, and LSI [44]. It has been shown that, for small wordlengths, the RNS can provide a significant speed-up using the $2^4 \times 2$ -bit tables found in Xilinx FPGAs [46]. For larger moduli, the M2K and M9K tables belonging to the Altera FPGAs are beneficial in designing RNS arithmetic and RNS-to-integer converters. With the ability to support larger moduli, the design of high-precision high-speed FPGA systems becomes a practical reality.

A historical barrier to implementing practical RNS systems, until recently, has been decoding [47]. Implementing RNS-to-integer decoder, division, or

magnitude scaling, requires that data first be converted from an RNS format to an integer. The commonly referenced RNS-to-integer conversion methods are called the Chinese remainder theorem (CRT) and the mixed-radix-conversion (MRC) algorithm [43]. The MRC actually produced the digits of a weighted number system representation of an integer while the CRT maps an RNS L -tuple directly to an integer. The CRT is defined below.

$$X \bmod M \equiv \sum_{l=0}^{L-1} \hat{m}_l \langle \hat{m}_l^{-1} x_l \rangle_{m_l} \bmod M, \quad (2.13)$$

where $\hat{m}_l = M/m_l$ is an integer, and \hat{m}_l^{-1} is the multiplicative inverse of $\hat{m}_l \bmod m_l$, i.e., $\hat{m}_l \hat{m}_l^{-1} \equiv 1 \bmod m_l$. Typically, the desired output of an RNS computation is much less than the maximum dynamic range M . In such cases, a highly efficient algorithm, called the ε -CRT [48], can be used to implement a time- and area-efficient RNS to (scaled) integer conversion.

Index Multiplier

There are, in fact, several variations of the RNS. One in common use is based on the use of index arithmetic [43]. It is similar in some respects to logarithmic arithmetic. Computation in the index domain is based on the fact that, if all the moduli are primes, it is known from number theory that there exists a primitive element, a *generator* g , such that:

$$a \equiv g^\alpha \bmod p \quad (2.14)$$

that generates all elements in the field \mathbb{Z}_p , excluding zero (denoted $\mathbb{Z}_p/\{0\}$). There is, in fact, a one-to-one correspondence between the integers a in $\mathbb{Z}_p/\{0\}$ and the exponents α in \mathbb{Z}_{p-1} . As a point of terminology, the index α , with respect to the generator g and integer a , is denoted $\alpha = \text{ind}_g(a)$.

Example 2.9: Index Coding

Consider a prime moduli $p = 17$; a generator $g = 3$ will generate the elements of $\mathbb{Z}_p/\{0\}$. The encoding table is shown below. For notational purposes, the case $a = 0$ is denoted by $g^{-\infty} = 0$.

a	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\text{ind}_3(a)$	$-\infty$	0	14	1	12	5	15	11	10	2	3	7	13	4	9	6	8

2.9

Multiplication of RNS numbers can be performed as follows:

- 1) Map a and b into the index domain, i.e., $a = g^\alpha$ and $b = g^\beta$

- 2)** Add the index values modulo $p - 1$, i.e., $\nu = (\alpha + \beta) \bmod (p - 1)$
3) Map the sum back to the original domain, i.e., $n = g^\nu$

If the data being processed is in index form, then only exponent addition mod($p - 1$) is required. This is illustrated by the following example.

Example 2.10: Index Multiplication

Consider the prime moduli $p = 17$, generator $g = 3$, and the results shown in Example 2.9. The multiplication of $a = 2$ and $b = 4$ proceeds as follows:

$$(\text{ind}_g(2) + \text{ind}_g(4)) \bmod 16 = (14 + 12) \bmod 16 = 10.$$

From the table in Example 2.9 it is seen that $\text{ind}_3(8) = 10$, which corresponds to the integer 8, which is the expected result. 2.10

Addition in the Index Domain

Most often, DSP algorithms require both multiplication *and* addition. Index arithmetic is well suited to multiplication, but addition is no longer trivial. Technically, addition can be performed by converting index RNS data back into the RNS where addition is simple to implement. Once the sum is computed the result is mapped back into the index domain. Another approach is based on a Zech logarithm. The sum of index-coded numbers a and b is expressed as:

$$d = a + b = g^\delta = g^\alpha + g^\beta = g^\alpha (1 + g^{\beta-\alpha}) = g^\beta (1 + g^{\alpha-\beta}). \quad (2.15)$$

If we now define the Zech logarithm as

Definition 2.11: Zech Logarithm

$$Z(n) = \text{ind}_g(1 + g^n) \longleftrightarrow g^{Z(n)} = 1 + g^n \quad (2.16)$$

then we can rewrite (2.15) in the following way:

$$g^\delta = g^\beta \times g^{Z(\alpha-\beta)} \longleftrightarrow \delta = \beta + Z(\alpha - \beta). \quad (2.17)$$

Adding numbers in the index domain, therefore, requires one addition, one subtraction, and a Zech LUT. The following small example illustrates the principle of adding 2 + 5 in the index domain.

Example 2.12: Zech Logarithms

A table of Zech logarithms, for a prime moduli 17 and $g = 3$, is shown below.

n	$-\infty$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$Z(n)$	0	14	12	3	7	9	15	8	13	$-\infty$	6	2	10	5	4	1	11

The index values for 2 and 5 are defined in the tables found in Example 2.9 (p. 71). It therefore follows that:

$$2 + 5 = 3^{14} + 3^5 = 3^5(1 + 3^9) = 3^{5+Z(9)} = 3^{11} \equiv 7 \pmod{17}.$$

2.12

The case where $a + b \equiv 0$ needs special attention, corresponding to the case where [49]:

$$-X \equiv Y \pmod{p} \iff g^{\alpha+(p-1)/2} \equiv g^\beta \pmod{p}.$$

That is, the sum is zero if, in the index domain, $\beta = \alpha + (p-1)/2 \pmod{(p-1)}$. An example follows.

Example 2.13: The addition of 5 and 12 in the original domain is given by $5 + 12 = 3^5 + 3^{13} = 3^5(1 + 3^8) = 3^{5+Z(8)} \equiv 3^{-\infty} \equiv 0 \pmod{17}$.

2.13

Complex Multiplication using QRNS

Another interesting property of the RNS arises if we process complex data. This special representation, called QRNS, allows very efficient multiplication, which we wish to discuss next.

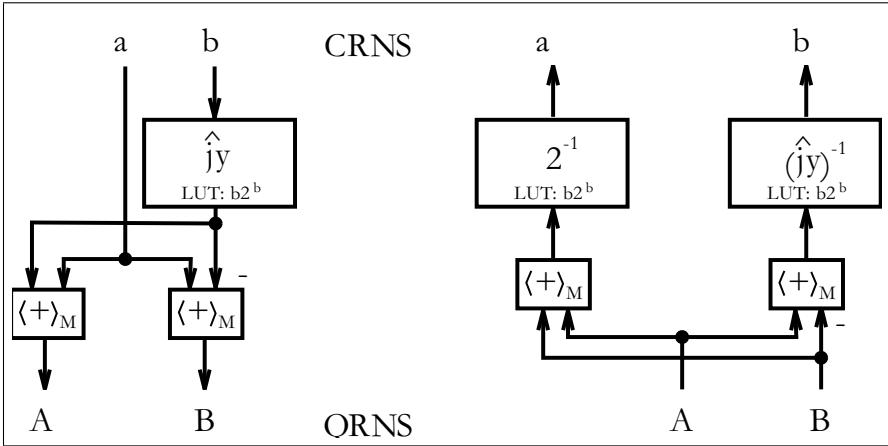
When the real and imaginary components are coded as RNS digits, the resulting system is called the complex RNS or CRNS. Complex addition in the CRNS requires that two real adds be performed. Complex RNS (CRNS) multiplication is defined in terms of four real products, an addition, and a subtraction. This condition is radically changed when using a variant of the RNS, called the quadratic RNS, or QRNS. The QRNS is based on known properties of Gaussian primes of the form $p = 4k + 1$, where k is a positive integer. The importance of this choice of moduli is found in the factorization of the polynomial $x^2 + 1$ in \mathbb{Z}_p . The polynomial has two roots, \hat{j} and $-\hat{j}$, where \hat{j} and $-\hat{j}$ are real integers belonging to the residue class \mathbb{Z}_p . This is in sharp contrast with the factoring of $x^2 + 1$ over the complex field. Here, the roots are complex and have the form $x_{1,2} = \alpha \pm j\beta$ where $j = \sqrt{-1}$ is the imaginary operator. Converting a CRNS number into the QRNS is accomplished by the transform $f : \mathbb{Z}_p^2 \rightarrow \mathbb{Z}_p^2$, defined as follows:

$$f(a + jb) = ((a + jb) \pmod{p}, (a - jb) \pmod{p}) = (A, B). \quad (2.18)$$

In the QRNS, addition and multiplication is realized componentwise, and is defined as

$$(a + ja) + (c + jd) \leftrightarrow (A + C, B + D) \pmod{p} \quad (2.19)$$

$$(a + jb)(c + jd) \leftrightarrow (AC, BD) \pmod{p} \quad (2.20)$$

**Fig. 2.6.** CRNS \leftrightarrow QRNS conversion

and the square of the absolute value can be computed with

$$|a + jb|^2 \leftrightarrow (A \times B) \mod p. \quad (2.21)$$

The inverse mapping from QRNS digits back to the CRNS is defined by:

$$f^{-1}(A, B) = 2^{-1}(A + B) + j(2\hat{j})^{-1}(A - B) \mod p. \quad (2.22)$$

Consider the Gaussian prime $p = 13$ and the complex product of $(a + jb) = (2 + j1)$, $(c + jd) = (3 + j2)$, is $(2 + j1) \times (3 + j2) = (4 + j7) \mod 13$. In this case four real multiplies, a real add, and real subtraction are required to complete the product.

Example 2.14: QRNS Multiplication

The quadratic equation $x^2 \equiv (-1) \mod 13$ has two roots: $\hat{j} = 5$ and $-\hat{j} = -5 \equiv 8 \mod 13$. The QRNS-coded data become:

$$\begin{aligned} (a + jb) &= 2 + j \leftrightarrow (2 + 5 \times 1, 2 + 8 \times 1) = (A, B) = (7, 10) \mod 13 \\ (c + jd) &= 3 + j2 \leftrightarrow (3 + 5 \times 2, 3 + 8 \times 2) = (C, D) = (0, 6) \mod 13. \end{aligned}$$

Componentwise multiplication yields $(A, B)(C, D) = (7, 10)(0, 6) \equiv (0, 8) \mod 13$, requiring only two real multiplies. The inverse mapping to the CRNS is defined in terms of (2.22), where $2^{-1} \equiv 7$ and $(2\hat{j})^{-1} = 10^{-1} \equiv 4$. Solving the equations for $2x \equiv 1 \mod 13$ and $10x \equiv 1 \mod 13$, produces 7 and 4, respectively. It then follows that

$$f^{-1}(0, 8) = 7(0 + 8) + j4(0 - 8) \mod 13 \equiv 4 + j7 \mod 13. \checkmark$$

2.14

Figure 2.6 shows a graphical interpretation of the mapping between CRNS and QRNS.

2.2.3 Floating-Point Numbers

Floating-point systems were developed to provide high resolution over a large dynamic range. Floating-point systems can often provide a solution when fixed-point systems, with their limited dynamic range, fail. Floating-point systems, however, bring a speed and complexity penalty. Most microprocessor floating-point systems comply with the published single- or double-precision IEEE floating-point standard [50], while FPGA-based systems often employ custom formats [51]. We will therefore discuss in the following the standard and custom floating-point formats, and in Sect. 2.7 (p. 109) the design of basic building blocks. Such arithmetic blocks are available from several “intellectual property” providers, and have recently been included in the VHDL-2008 standard.

A standard normalized floating-point word consists of a sign-bit s , exponent e , and an unsigned (fractional) normalized mantissa m , arranged as follows:

Sign s	Exponent e	Unsigned mantissa m
----------	--------------	-----------------------

Algebraically, a (normalized) floating-point word is represented by

$$X = (-1)^s \times 1.m \times 2^{e-\text{bias}}. \quad (2.23)$$

Note that this is a signed magnitude format (see p. 59). The “hidden” one in the mantissa is not present in the binary coding of the normalized floating-point number. If the exponent is represented with E bits then the bias is selected to be

$$\text{bias} = 2^{E-1} - 1. \quad (2.24)$$

To illustrate, let us determine the decimal value 9.25 in a 12-bit custom floating-point format.

Example 2.15: A (1,6,5) Floating-Point Format

Consider a floating-point representation with a sign bit, $E = 6$ -bit exponent width, and $M = 5$ -bit for the mantissa (not counting the hidden one). Let us now determine the representation of 9.25_{10} in this (1,6,5) floating-point format. Using (2.24) the bias is

$$\text{bias} = 2^{E-1} - 1 = 31,$$

and the mantissa needs to be normalized according to the $1.m$ format, i.e.,

$$9.25_{10} = 1001.01_2 = 1.\underbrace{00101}_m \times 2^3.$$

The biased exponent is therefore represented with

$$e = 3 + \text{bias} = 34_{10} = 100010_2.$$

Finally, we can represent 9.25_{10} in the (1,6,5) floating-point format with

Sign s	Exponent e	Unsigned mantissa m
0	100010	00101

Besides this fixed-point to floating-point conversion we also need the back conversion from floating-point to fixed-point or integer. So, let us assume the floating-point number

s	Exponent e	Unsigned mantissa m
1	011111	00000

is given and we wish to find the fixed-point representation of this number. We first notice that the sign bit is one, i.e., it is a negative number. Adding the hidden one to the mantissa and subtracting the bias from the exponent, yields

$$-1.00000_2 \times 2^{31-\text{bias}} = -1.0_2 2^0 = -1.0_{10}.$$

We note that in the floating-point to fixed-point conversion the bias is subtracted from the exponent, while in the fixed-point to floating-point conversion the bias is added to the exponent.

2.15

The IEEE standard 754-2008 for binary floating-point arithmetic [51] also defines some additional useful special numbers to handle, for instance, overflow and underflow. The exponent $e = E_{\max} = 1\dots1_2$ in combination with zero mantissa $m = 0$ is reserved for ∞ . Zeros are coded with zero exponent $e = E_{\min} = 0$ and zero mantissa $m = 0$. Note that due to the signed magnitude representation, plus and minus zero are coded differently. There are two more special numbers defined in the 754 IEEE standard, but these additional representations are most often not supported in FPGA floating-point arithmetic. These additional number are *denormals* and *NaNs* (not a number). With denormalized numbers we can represent numbers smaller than $2^{E_{\min}}$ by allowing the mantissa to represent numbers without the hidden one, i.e., the mantissa can represents numbers smaller than 1.0. The exponent in denormals is coded with $e = E_{\min} = 0$, but the mantissa is allowed to be different from zero. NaNs have proven useful in software systems to reduce the number of “exceptions” that are called when an invalid operation is performed. Examples that produce such “quiet” NaNs include:

- Addition or subtraction of two infinities, such as $\infty - \infty$
- Multiplication of zero and infinite, e.g., $0 \times \infty$
- Division of zeros or infinities, e.g., $0/0$ or ∞/∞
- Square root of negative operand

In the IEEE standard 754 for binary floating-point arithmetic NaNs are coded with exponent $e = E_{\max} = 1\dots1_2$ in combination with a nonzero mantissa $m \neq 0$. Table 2.4 shows the five major floating-point codings including the special numbers.

We wish now to compare the fixed-point and floating-point representation in terms of precision and dynamic range in the following example.

Table 2.4. The five major coding types in the 754-1985 and updated 754-2008 IEEE binary floating point standard

Sign s	Exponent e	Mantissa m	Meaning
0/1	All-zeros	All-zeros	± 0
0/1	All-zeros	Nonzero	Denormalized: $(-1)^s 2^{-\text{Bias}} 0.m$
0/1	$1 < e < E_{\max}$	m	Normalized: $(-1)^s 2^{e-\text{Bias}} 1.m$
0/1	All-ones	All-zeros	$\pm \infty$
-	All-ones	Nonzero	NaN

Example 2.16: 12-Bit Floating- and Fixed-Point Representations

Suppose we use again a (1,6,5) floating-point format as in the previous example. The (absolute) largest number we can represent is

$$\pm 1.11111_2 \times 2^{31} \approx \pm 4.23_{10} \times 10^9.$$

The (absolutely measured) smallest number (not including denormals) that can be represented is

$$\pm 1.0_2 \times 2^{1-\text{bias}} = \pm 1.0_2 \times 2^{-30} \approx \pm 9.31_{10} \times 10^{-10}.$$

If we also allow denormalized numbers then the smallest number that can be represented becomes

$$\pm 0.00001_2 \times 2^{-\text{bias}} = \pm 1.0_2 \times 2^{-31-5} \approx \pm 1.45_{10} \times 10^{-11},$$

i.e., a factor 64 smaller than the normalized smallest number. Note that $e=0$ is reserved for the denormalized in the floating-point format; see Table 2.4. For the 12-bit fixed-point format we use one sign bit, five integer bits, and six fractional bits. The maximum (absolute) value we can represent with this 12-bit fixed-point format is therefore

$$\begin{aligned} \pm 11111.11111_2 &= \pm (16 + 8 + \dots + \frac{1}{32} + \frac{1}{64})_{10} \\ &= \pm (32 - \frac{1}{64})_{10} \approx \pm 32.0_{10}. \end{aligned}$$

The (absolutely measured) smallest number that this 12-bit fixed-point format represents is

$$\pm 00000.000001_2 = \pm \frac{1}{64}_{10} = \pm 0.015625_{10}.$$

2.16

From this example we notice the larger *dynamic range* of the floating-point representation (4×10^9 compared with 32 for the fixed-point) but also a higher *precision* of the fixed-point representation. For instance, 1.0 and $1 + 1/64 = 1.015625$ are coded the same in (1,6,5) floating-point format, but can be distinguished in 12-bit fixed-point representation.

There are two rounding mode for fixed-point type and four supported rounding modes for floating-point type which are rounding-to-nearest-even

Table 2.5. Rounding examples for the four floating-point types

Mode	32.5	33.25	33.5	33.75	-32.5	-32.25
Rounding-to-nearest-even	32	33	34	34	-32	-32
Rounding-to-zero	32	33	33	33	-32	-32
Rounding-to- ∞	33	34	34	34	-32	-32
Round-to-negative- ∞	32	33	33	33	-33	-33

(i.e., the default), rounding-to-zero (truncation), rounding-to- ∞ (round up), and round-to-negative- ∞ (round down). In MATLAB the equivalent rounding functions are `round()`, `fix()`, `ceil()`, and `floor()`, respectively. The only small difference between the MATLAB and IEEE 754 modes is the rounding-to-even-nearest for numbers with $0.5_{10} = 0.1_2$ fractional part. Only if the integer LSB is one do we round up – otherwise we round down; 32.5 is rounded down but 33.5 is rounded up in the rounding-to-even-nearest scheme. Table 2.5 shows an example rounding that may occur in the (1,6,5) floating-point format. It is interesting to observe that the default operation rounding-to-nearest-even is the most complicated scheme to implement and the rounding-to-zero is not only the cheapest but also may be used to reduce an undesired gain in the processing since we always round to zero, i.e., the amplitude does not grow due to rounding.

Although the IEEE standard 754-1985 for binary floating-point arithmetic [50] is not easy to implement with all its details, such as four different rounding modes, denormals, or NaNs, the early introduction in 1985 of the standard helped as it has become the most adopted implementation for microprocessors. The parameters of this IEEE single and double format can be seen from Table 2.6. Due to the fact that already single-precision 754 standard arithmetic designs will require

- a 24×24 -bit multiplier, and
- FPGAs allow a more specific dynamic range design (i.e., exponent bit width) and precision (mantissa bit width) design

we find that sometimes FPGA designer do not adopt the IEEE 754 standard and define a special format. Shirazi et al. [52], for instance, have developed a modified format to implement various algorithms on their custom computing machine called SPLASH-2, a multiple-FPGA board based on Xilinx XC4010 devices. They used an 18-bit format so that they can transport two operands over the 36-bit wide system bus of the multiple-FPGA board. The 18-bit format has a 10-bit mantissa, 7-bit exponent and a sign bit, and can represent a range of 3.7×10^{19} .

Table 2.6. IEEE floating-point 754-2008 standard interchange formats

	Short	Single	Double	Extended
Word length	16	32	64	128
Mantissa	10	23	52	112
Exponent	5	8	11	15
Bias	15	127	1023	16383
Range	$2^{16} \approx 6.4 \times 10^4$	$2^{128} \approx 3.8 \times 10^{38}$	$2^{1024} \approx 1.8 \times 10^{308}$	$2^{16384} \approx 10^{4932}$

2.3 Binary Adders

A basic binary N -bit adder/subtractor consists of N full-adders (FA). A full-adder implements the following Boolean equations:

$$s_k = x_k \text{ XOR } y_k \text{ XOR } c_k \quad (2.25)$$

$$= x_k \oplus y_k \oplus c_k \quad (2.26)$$

that define the sum-bit. The carry (out) bit is computed with

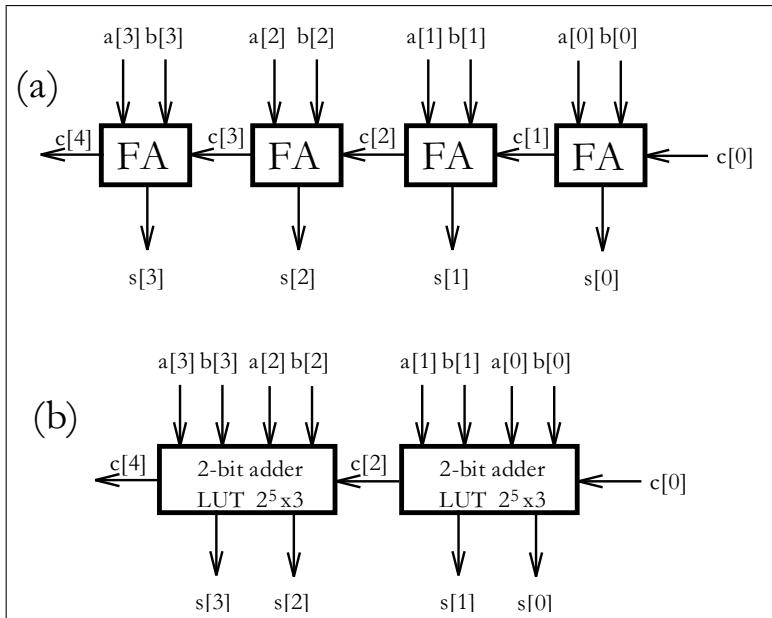
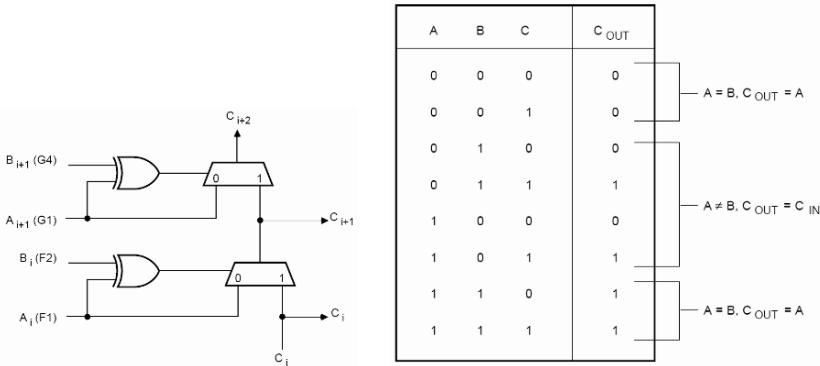
$$c_{k+1} = (x_k \text{ AND } y_k) \text{ OR } (x_k \text{ AND } c_k) \text{ OR } (y_k \text{ AND } c_k) \quad (2.27)$$

$$= (x_k \times y_k) + (x_k \times c_k) + (y_k \times c_k) \quad (2.28)$$

In the case of a 2C adder, the LSB can be reduced to a half-adder because the carry input is zero.

The simplest adder structure is called the “ripple carry adder” as shown in Fig. 2.7a in a bit-serial form. If larger tables are available in the FPGA, several bits can be grouped together into one LUT, as shown in Fig. 2.7b. For this “two bit at a time” adder the longest delay comes from the ripple of the carry through all stages. Attempts have been made to reduce the carry delays using techniques such as the carry-skip, carry lookahead, conditional sum, or carry-select adders. These techniques can speed up addition and can be used with older-generation FPGA families (e.g., XC 3000 from Xilinx) since these devices do not provide internal fast carry logic. Modern families, such as the Xilinx Spartan or Altera Cyclone, possess very fast “ripple carry logic” that is about a magnitude faster than the delay through a regular logic LUT [1]. Altera uses fast tables (see Fig. 1.12, p. 23), while the Xilinx uses hardwired decoders for implementing carry logic based on the multiplexer structure shown in Fig. 2.8; see also Fig. 1.11, p. 21. The presence of the fast-carry logic in modern FPGA families removes the need to develop hardware intensive carry look-ahead schemes.

Figure 2.9 summarizes the size and registered performance of N -bit binary adders, if implemented with the `lpm_add_sub` megafunction component. Beside the EP4CE115F29C7 from the Cyclone IV E family (that is built currently using 60-nm process technology), we have also included as a reference the data for mature families. The EP20K200EFC484-2X is from the

**Fig. 2.7.** Two's complement adders**Fig. 2.8.** XC4000 fast-carry logic. (© Xilinx [53])

APEX20KE family and can be found on the Nios development boards, see Chap. 9. The APEX20KE family was introduced in 1999 and used a $0.18\mu m$ process technology. The EPF10K70RC240-4 is from the FLEX10K family and can be found on the UP2 development boards. The FLEX10K family was introduced in 1995 and used a $0.42\mu m$ process technology. Although the LE cell structure has not changed much over time we can see from the advance in process technology the improvement in speed. If the operands are placed in I/O register cells, the delays through the busses of a FPGA are

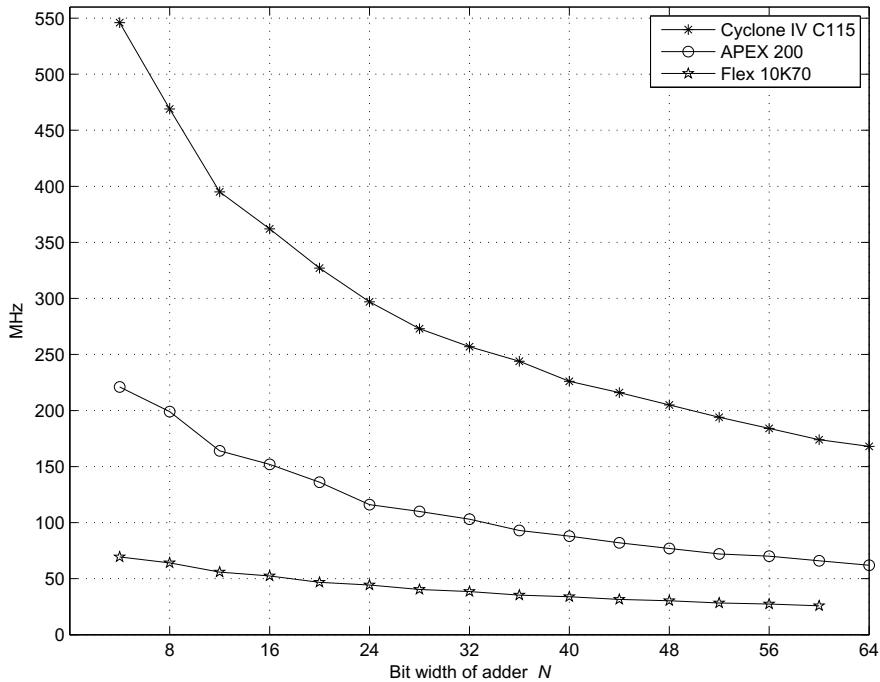


Fig. 2.9. Adder speed for Cyclone IV, APEX, and Flex10K

dominant and performance decreases. If the data are routed from local registers, performance improves. For this type of design additional LE register allocation will appear (in the project report file) as increased LE use by a factor of three or four. However, a synchronous registered larger system would not consume any additional resources since the data are registered at the previous processing stage. A typical design will achieve a speed between these two cases. For Flex10K the adder and register are not merged, and $4 \times N$ LEs are required. LE requirements for the Cyclone IV and APEX devices are $3 \times N$ for the speed data shown in Fig. 2.9.

2.3.1 Pipelined Adders

Pipelining is extensively used in DSP solutions due to the intrinsic dataflow regularity of DSP algorithms. Programmable digital signal processor MACs [6, 16, 17] typically carry at least four pipelined stages. The processor:

- 1) Decodes the command
- 2) Loads the operands in registers
- 3) Performs multiplication and stores the product, and
- 4) Accumulates the products, all concurrently.

Table 2.7. Performance of a 14-bit pipelined adder for the EP2C35F672C6 using synthesis of predefined LPM modules with pipeline option

Pipeline stages	MHz	LEs
0	375.94	42
1	377.79	56
2	377.64	70
3	377.79	84
4	381.10	98
5	376.36	112

The pipelining principle can be applied to FPGA designs as well, at little or no additional cost since each logic element contains a flip-flop, which is otherwise unused, to save routing resources. With pipelining it is possible to break an arithmetic operation into small primitive operations, save the carry and the intermediate values in registers, and continue the calculation in the next clock cycle. Such adders are sometimes called carry save adders⁴ (CSAs) in the literature. Then the question arises: In how many pieces should we divide the adder? Should we use bit level? For Altera's Cyclone IV devices a reasonable choice will be always using an LAB with 16 LEs and 16 FFs for one pipeline element. The FLEX10K family has 8 LEs per LAB, while APEX20KE uses 10 LEs per LAB. So we need to consult the datasheet before we make a decision on the size of the pipelining group. In fact, it can be shown that if we try to pipeline (for instance) a 14-bit adder in our Cyclone IV devices, the performance does not improve, as reported in Table 2.7, because the pipelined 14-bit adder does not fit in one LAB.

Because the number of flip-flops in one LAB is 16 and we need an extra flip-flop for the carry-out, we should use a maximum block size of 15 bits for maximum registered performance. Only the blocks with the MSBs can be 16 bits wide, because we do not need the extra flip-flop for the carry. This observation leads to the following conclusions:

- 1) With one additional pipeline stage we can build adders up to a length $15 + 16 = 31$.
- 2) With two pipeline stages we can build adders with up to $15 + 15 + 16 = 46$ -bit length.
- 3) With three pipeline stages we can build adders with up to $15 + 15 + 15 + 16 = 61$ -bit length.

Table 2.8 shows the registered performance and LE utilization of this kind of pipelined adder. From Table 2.8 it can be concluded that although the

⁴ The name carry save adder is also used in the context of a Wallace multiplier, see Exercise 2.1, p. 168.

Table 2.8. Performance and resource requirements of adders with and without pipelining. Size and speed are for the maximum bit width, for 31-, 46-, and 61-bit adders

Bit width	No pipeline		With pipeline		Pipeline stages	Design file name
	MHz	LEs	MHz	LEs		
17 – 31	263.50	93	350.63	125	1	add1p.vhd
32 – 46	215.66	138	243.43	233	2	add2p.vhd
47 – 61	173.13	183	231.43	372	3	add3p.vhd

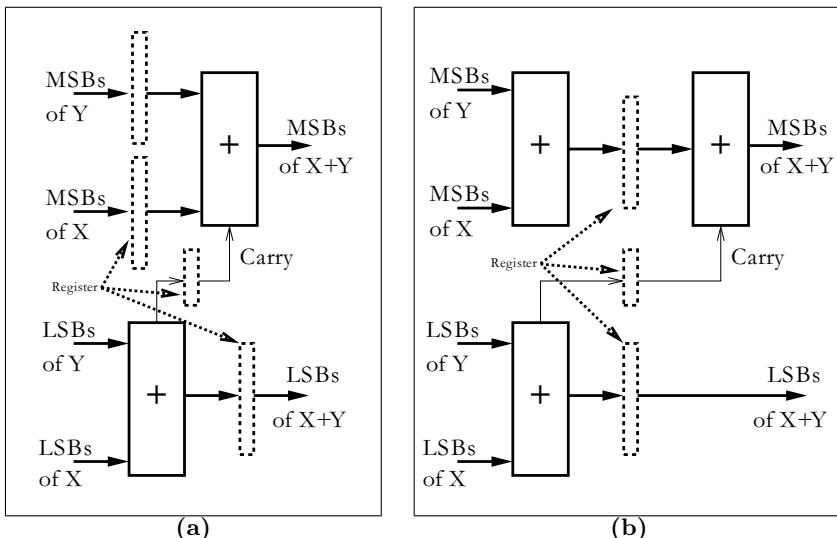


Fig. 2.10. Pipelined adder. (a) Direct implementation. (b) FPGA optimized approach

bit width increases the registered performance remains high if we add the appropriate number of pipeline stages.

The following example shows the code of a 31-bit pipelined adder. It turns out that the straight forward implementation of the pipelining would require two registers for the MSBs as shown in Fig. 2.10a. If we instead use adders for the MSBs, we can save a set of LEs, since each LE can implement a full adder, but only one flip-flop. This is graphically interpreted by Fig. 2.10b.

Example 2.17: VHDL Design of 31-bit Pipelined Adder

Consider the VHDL code⁵ of a 31-bit pipelined adder that is graphically interpreted in Fig. 2.10. The design runs at 350.63 MHz and uses 125 LEs.

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
-----
ENTITY add1p IS
    GENERIC (WIDTH  : INTEGER := 31; -- Total bit width
              WIDTH1 : INTEGER := 15; -- Bit width of LSBs
              WIDTH2 : INTEGER := 16); -- Bit width of MSBs
    PORT (x,y : IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
          -- Inputs
          sum : OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
          -- Result
          LSBs_carry : OUT STD_LOGIC; -- Test port
          clk : IN STD_LOGIC); -- System clock
END add1p;
-----
ARCHITECTURE fpga OF add1p IS
    SIGNAL l1, l2, s1           -- LSBs of inputs
                                : STD_LOGIC_VECTOR(WIDTH1-1 DOWNTO 0);
    SIGNAL r1                   -- LSBs of inputs
                                : STD_LOGIC_VECTOR(WIDTH1 DOWNTO 0);
    SIGNAL l3, l4, r2, s2       -- MSBs of inputs
                                : STD_LOGIC_VECTOR(WIDTH2-1 DOWNTO 0);
    BEGIN
        PROCESS -- Split in MSBs and LSBs and store in registers
        BEGIN
            WAIT UNTIL clk = '1';
            -- Split LSBs from input x,y
            l1 <= x(WIDTH1-1 DOWNTO 0);
            l2 <= y(WIDTH1-1 DOWNTO 0);
            -- Split MSBs from input x,y
            l3 <= x(WIDTH-1 DOWNTO WIDTH1);
            l4 <= y(WIDTH-1 DOWNTO WIDTH1);
            ----- First stage of the adder -----
            r1 <= ('0' & l1) + ('0' & l2);
            r2 <= l3 + l4;
            ----- Second stage of the adder -----
            s1 <= r1(WIDTH1-1 DOWNTO 0);
            -- Add result von MSBs (x+y) and carry from LSBs
            s2 <= r1(WIDTH1) + r2;
        END PROCESS;
        LSBs_Carry <= r1(WIDTH1); -- Add a test signal
        -- Build a single output word of WIDTH=WIDTH1+WIDTH2
    
```

⁵ The equivalent Verilog code `add1p.v` for this example can be found in Appendix A on page 797. Synthesis results are shown in Appendix B on page 881.

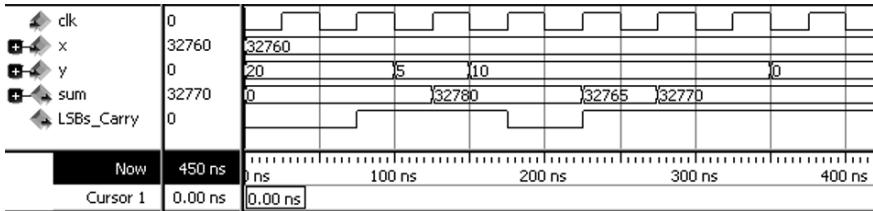


Fig. 2.11. Simulation results for a pipelined adder

```
sum <= s2 & s1 ; -- Connect s to output pins
```

```
END fpga;
```

The simulated performance of the 15-bit pipelined adder shows Fig. 2.11. Note that the addition results for 32780 and 32770 produce a carry from the lower 15-bit adder, but there is no carry for $32760 + 5 = 32765 < 2^{15}$. [2.17]

2.3.2 Modulo Adders

Modulo adders are the most important building blocks in RNS-DSP designs. They are used for both additions and, via index arithmetic, for multiplications. We wish to describe some design options for FPGAs in the following discussion.

A wide variety of *modular* addition designs exists [54]. Using LEs only, the design of Fig. 2.12a is viable for FPGAs. The Altera FLEX devices contain a small number of M2K ROMs or RAMs (EABs) that can be configured as $2^8 \times 8$, $2^9 \times 4$, $2^{10} \times 2$ or $2^{11} \times 1$ tables and can be used for modulo m_l correction. The next table shows size and registered performance 6, 7, and 8-bit modulo adder compile for Altera FLEX10K devices [55].

	Pipeline stages	Bits		
		6	7	8
MPX	0	41.3 MSPS 27 LE	46.5 MSPS 31 LE	33.7 MSPS 35 LE
	2	76.3 MSPS 16 LE	62.5 MSPS 18 LE	60.9 MSPS 20 LE
MPX	3	151.5 MSPS 27 LE	138.9 MSPS 31 LE	123.5 MSPS 35 LE
		86.2 MSPS	86.2 MSPS	86.2 MSPS
	3	7 LE 1 EAB	8 LE 1 EAB	9 LE 2 EAB
ROM				

Although the ROM shown in Fig. 2.12 provides high speed, the ROM itself produces a four-cycle pipeline delay and the number of ROMs is limited.

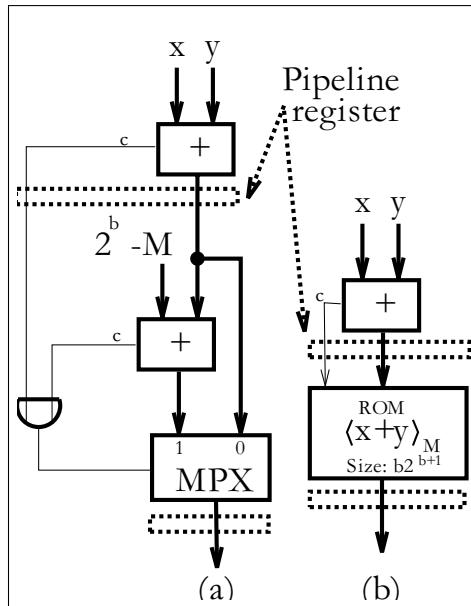


Fig. 2.12. Modular additions. (a) MPX-Add and MPX-Add-Pipe. (b) ROM-Pipe

ROMs, however, are mandatory for the scaling schemes discussed before. The multiplexed-adder (MPX-Add) has a comparatively reduced speed even if a carry chain is added to each column. The pipelined version usually needs the same number of LEs as the unpipelined version but runs about three times as fast. Maximum throughput occurs when the adders are implemented with 3 pipeline stages and 6-bit width channels.

2.4 Binary Multipliers

The product of two N -bit binary numbers, say X and $A = \sum_{k=0}^{N-1} a_k 2^k$, is given by the “pencil and paper” method as:

$$P = A \times X = \sum_{k=0}^{N-1} a_k 2^k X. \quad (2.29)$$

It can be seen that the input X is successively shifted by k positions and whenever $a_k \neq 0$, then $X2^k$ is accumulated. If $a_k = 0$, then the corresponding shift-add can be ignored (i.e., nop). With the introduction of embedded multipliers in recent FPGAs this FSM approach is not used often.

Because one operand is used in parallel (i.e., X) and the second operand A is used bitwise, the multipliers we just described are called serial/parallel multipliers. If both operands are used serial, the scheme is called a serial/serial

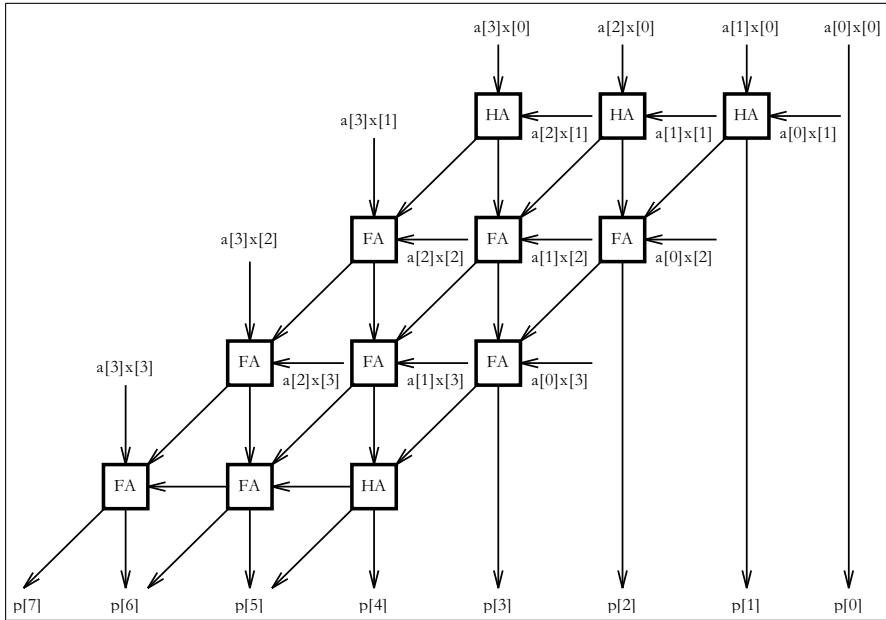


Fig. 2.13. A 4-bit array multiplier

multiplier [56], and such a multiplier only needs one full adder, but the latency of serial/serial multipliers is high $\mathcal{O}(N^2)$, because the state machine needs about N^2 cycles.

Another approach, which trades speed for increased complexity, is called an “array,” or parallel/parallel multiplier. A 4×4 -bit array multiplier is shown in Fig. 2.13. Notice that both operands are presented in parallel to an adder array of N^2 adder cells.

This arrangement is viable if the times required to complete the carry and sum calculations are the same. For a modern FPGA, however, the carry computation is performed faster than the sum calculation and a different architecture is more efficient for FPGAs. The approach for this array multiplier is shown in Fig. 2.14, for an 8×8 -bit multiplier. This scheme combines in the first stage two neighboring partial products $a_n X 2^n$ and $a_{n+1} X 2^{n+1}$ and the results are added to arrive at the final output product. This is a direct array form of the “pencil and paper” method and must therefore produce a valid product.

We recognize from Fig. 2.14 that this type of array multiplier gives the opportunity to realize a (parallel) *binary tree* of the multiplier with a total:

$$\text{number of stages in the binary tree multiplier} = \log_2(N). \quad (2.30)$$

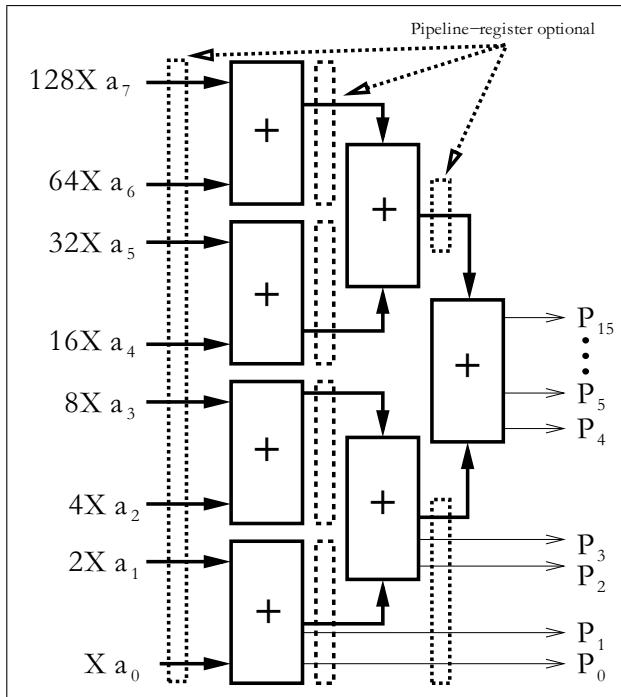


Fig. 2.14. Fast array multiplier for FPGAs

This alternative architecture also makes it easier to introduce pipeline stages after each tree level. The necessary number of pipeline stages, according to (2.30), to achieve maximum throughput is:

Bit width	2	3 – 4	5 – 8	9 – 16	17 – 32
Optimal number of pipeline stages	1	2	3	4	5

Since the data are registered at the input and output the number of delays in the simulation would be two larger than the pipeline stage we specified for the `1pm_mul` blocks.

Figure 2.15 reports the `Fmax` registered performance of pipelined $N \times N$ -bit multipliers, using the Quartus II `1pm_mult` function, for 8×8 , to 24×24 bits operands. Embedded multiplier are shown with dash lines and up to 16×16 -bit the multiplier do not improve with pipelining since they fit in one embedded 18×18 -bit array multiplier. The LE-based multiplier are shown with a solid line. Figure 2.16 shows the LEs effort for the multiplier. The pipelined 8×8 bit multiplier outperforms the embedded multiplier if 2 or more pipeline stages are used. We can notice from Fig. 2.15 that, for pipeline delays longer than $\log_2(N)$, there is no essential improvement for LE-based multi-

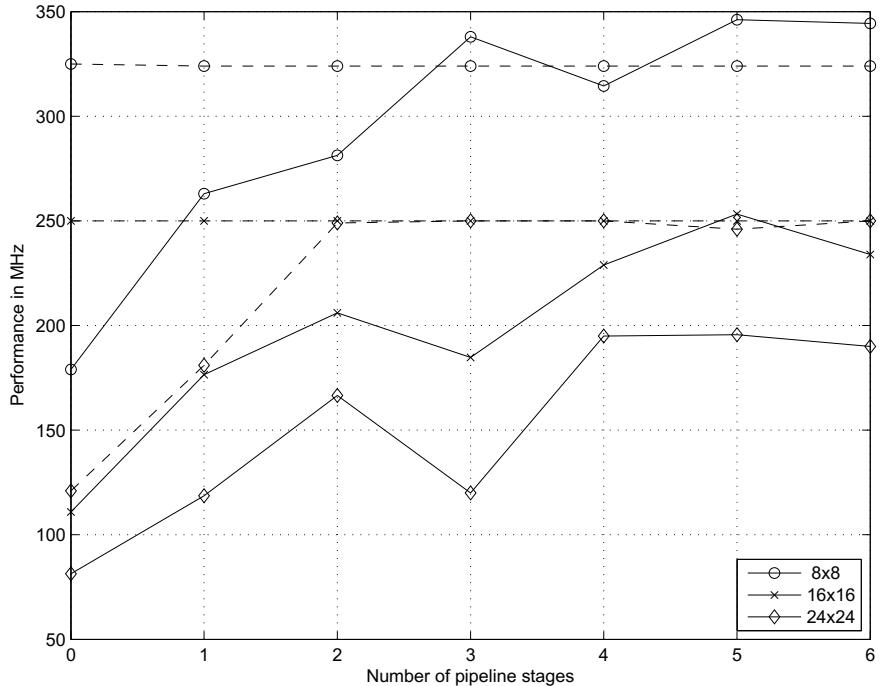


Fig. 2.15. Performance of an array multiplier for FPGAs, LE-based multiplier (solid line) and embedded multiplier (dashed line)

pliers. The multiplier architecture (embedded or LEs) must be controlled via synthesis options in case we write behavioral code (e.g., $p \leq a*b$). This can be done under **Setting** in the **Assignments** menu. There you find the **DSP Block Balancing** entry under the **Analysis & Synthesis Settings** → **More Settings**. Select **DSP blocks** if you like to use the embedded multiplier, **Logic Elements** to use the LEs only, or **Auto**, and the synthesis tool will first use the embedded multiplier; if there are not enough then use the LE-based multiplier. If we use the **lpm_mul** block we have direct control using the **GENERIC MAP** parameter **DEDICATED_MULTIPLIER_CIRCUITRY => "YES"** or **"NO"**.

Other multiplier architectures typically used in the ASIC world include Wallace-tree multipliers and Booth multipliers. They are discussed in Exercises 2.1 (p. 168) and 2.2 (p. 169) but are rarely used in connection with FPGAs.

2.4.1 Multiplier Blocks

A $2N \times 2N$ multiplier can be defined in terms of an $N \times N$ multiplier block [33]. The resulting multiplication is defined as:

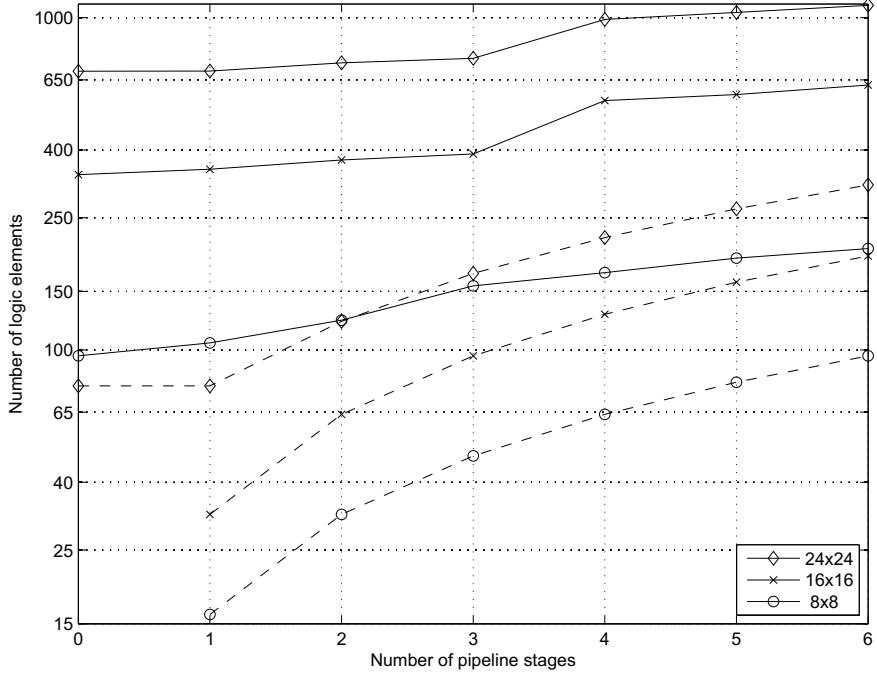


Fig. 2.16. Effort in LEs for array multipliers, LE-based multiplier (*solid line*) and embedded multiplier (*dashed line*)

$$P = Y \times X = (Y_2 2^N + Y_1)(X_2 2^N + X_1) \quad (2.31)$$

$$= Y_2 X_2 2^{2N} + (Y_2 X_1 + Y_1 X_2) 2^N + Y_1 X_1, \quad (2.32)$$

where the indices 2 and 1 indicate the most significant and least significant N -bit halves, respectively. This partitioning scheme can be used if the capacity of the FPGA is insufficient to implement a multiplier of desired size, or used to implement a multiplier using memory blocks. A 36×36 -bit multiplier can be built with four 18×18 -bit embedded multipliers and three adders. An 8×8 -bit LUT-based multiplier in direct form would require an LUT size of $2^{16} \times 16 = 1$ Mbit. The partitioning technique reduces the table size to four $2^8 \times 8$ memory blocks and three adders. A 16×16 -bit multiplier requires 16 memory blocks. The benefit of multiplier implementation via M9Ks versus LE-based is twofold. First, the number of LE is reduced. Secondly, the requirements on the routing resources of the devices are also reduced. Although some FPGAs families now have a limited number of embedded array multipliers, the number is usually small, and the LUT-based multiplier provides a way to enlarge the number of fast low-latency multipliers in these devices. In addition, some device families like Cyclone, Flex, or Excalibur do not have

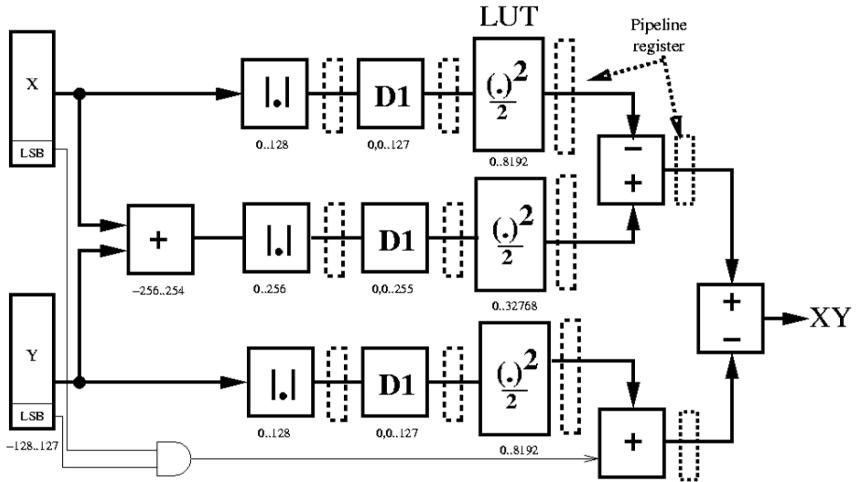


Fig. 2.17. Two's complement 8-bit additive half-square multiplier design

embedded multipliers; therefore, the LUT or LE multipliers are the only option.

Half-Square Multiplier

Another way to reduce the memory requirement for LUT-based multipliers is to decrease the bits in the input domain. One bit decrease in the input domain decreases the number of LUT words by a factor of two. An LUT of a square operation of an N -bit word only requires an LUT size of $2^N \times 2^N$. The additive half-square (AHSM) multiplier

$$Y \times X = \frac{(X + Y)^2 - X^2 - Y^2}{2} = \\ = \left\lfloor \frac{(X + Y)^2}{2} \right\rfloor - \left\lfloor \frac{X^2}{2} \right\rfloor - \left\lfloor \frac{Y^2}{2} \right\rfloor - \begin{cases} 1 & X, Y \text{ odd} \\ 0 & \text{others} \end{cases} \quad (2.33)$$

was introduced by Logan [57]. If the division by 2 is included in the LUT, this requires a correction of -1 in the event that X and Y are odd. A differential half-square multiplier (DHSM) can then be implemented as:

$$Y \times X = \frac{(X + Y)^2 - X^2 - Y^2}{2} = \\ = \left\lfloor \frac{X^2}{2} \right\rfloor + \left\lfloor \frac{Y^2}{2} \right\rfloor - \left\lfloor \frac{(X - Y)^2}{2} \right\rfloor + \begin{cases} 1 & X, Y \text{ odd} \\ 0 & \text{others} \end{cases}. \quad (2.34)$$

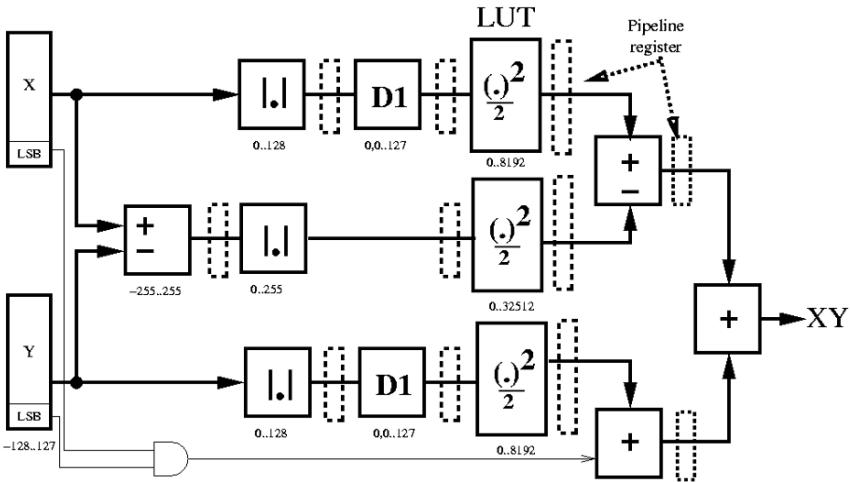


Fig. 2.18. Two's complement 8-bit differential half-square multiplier design

A correction of 1 is required in the event that X and Y are odd. If the numbers are signed, an additional saving is possible by using the diminished-by-one (D1) encoding, see Sect. 2.2.1, p. 60. In D1 coding all numbers are diminished by 1, and the zero gets special encoding [58]. Figure 2.17 shows for 8-bit data the AHSM multiplier, the required LUTs, and the data range of 8-bit input operands. The absolute operation almost allows a reduction by a factor of 2 in LUT words, while the D1 encoding enables a reduction to the next power-of-two table size that is beneficial for the FPGA design. Since LUT inputs 0 and 1 both have the same square, LUT entry $\lfloor A^2/2 \rfloor$, we share this value and do not need to use special encoding for zero. Without the division by 2, a 17-bit output word would be required. However, the division by two in the squaring table requires an increment (decrement) of the output result for the AHSM (DHSM) in case both input operands are odd values. Figure 2.18 shows a DHSM multiplier that only requires two D1 encoding compared with the AHSM design.

Quarter-Square Multiplier

A further reduction in arithmetic requirements and the number of LUTs can be achieved by using the quarter-square multiplication (QSM) principle that is also well studied in analog designs [59, 60]. The QSM is based on the following equation:

$$Y \times X = \left\lfloor \frac{(X + Y)^2}{4} \right\rfloor - \left\lfloor \frac{(X - Y)^2}{4} \right\rfloor.$$

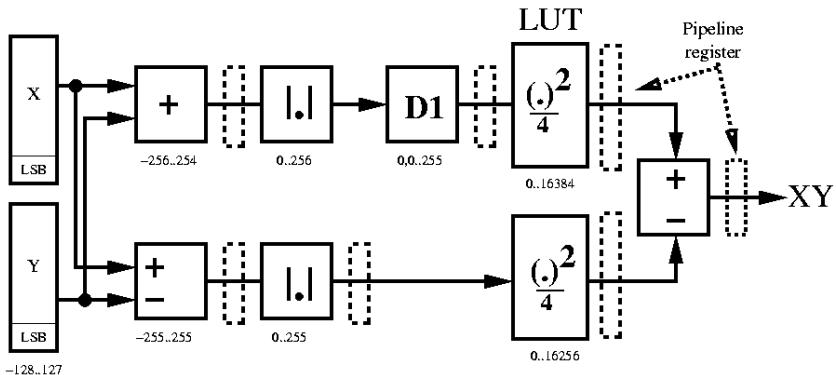


Fig. 2.19. Two's complement 8-bit quarter-square multiplier design

It is interesting to note that the division by 4 in (2.35) does not require any correction for operation as in the HSM case. This can be checked as follows. If both operands are even (odd), then the sum and the difference are both even, and the squaring followed by a division of 4 produces no error (i.e., $4|(2u * 2v)$). If one operand is odd (even) and the other operand is even (odd), then the sum and the difference after squaring and a division by 4 produce a 0.25 error that is annihilated in both cases. No correction operation is necessary. The direct implementation of (2.35) would require LUTs of $(N + 1)$ -bit inputs to represent the correct result of $X \pm Y$ as used in [61], which will require four $2^N \times 2^N$ LUTs. Signed arithmetic along with D1 coding will reduce the table to the next power-of-two value, allowing the design to use only two $2^N \times 2^N$ LUTs compared with the four in [61]. Figure 2.19 shows the D1 QSM circuit.

Synthesis results for AHSM, DHSM, and QSM can be found in [34].

2.5 Binary Dividers

Of all four basic arithmetic operations division is the most complex. Consequently, it is the most time-consuming operation and also the operation with the largest number of different algorithms to be implemented. For a given dividend (or numerator) N and divisor (or denominator) D the division produces (unlike the other basic arithmetic operations) two results: the quotient Q and the remainder R , i.e.,

$$\frac{N}{D} = Q \quad \text{and} \quad R \quad \text{with } |R| < D. \quad (2.35)$$

However, we may think of division as the inverse process of multiplication, as demonstrated through the following equation,

$$N = D \times Q + R, \quad (2.36)$$

it differs from multiplication in many aspects. Most importantly, in multiplication all partial products can be produced parallel, while in division each quotient bit is determined in a sequential “trail-and-error” procedure.

Because most microprocessors handle division as the inverse process to multiplications, referring to (2.36), the numerator is assumed to be the result of a multiplication and has therefore twice the bit width of denominator and quotient. As a consequence, the quotient has to be checked in an awkward procedure to be in the valid range, i.e., that there is no overflow in the quotient. We wish to use a more general approach in which we assume that

$$Q \leq N \quad \text{and} \quad |R| \leq D,$$

i.e., quotient and numerator as well as denominator and remainder are assumed to be of the same bit width. With this bit width assumptions no range check (except $N = 0$) for a valid quotient is necessary.

Another consideration when implementing division comes when we deal with signed numbers. Obviously, the easiest way to handle signed numbers is first to convert both to unsigned numbers and compute the sign of the result as an XOR or modulo 2 add operation of the sign bits of the two operands. But some algorithms, (like the nonrestoring division discussed below), can directly process signed numbers. Then the question arises, how are the sign of quotient and remainder related. In most hardware or software systems (but not for all, such as in the PASCAL programming language), it is assumed that the remainder and the quotient have the same sign. That is, although

$$\frac{234}{50} = 5 \quad \text{and} \quad R = -16 \quad (2.37)$$

meets the requirements from (2.36), we, in general, would prefer the following results:

$$\frac{234}{50} = 4 \quad \text{and} \quad R = 34. \quad (2.38)$$

Let us now start with a brief overview of the most commonly used division algorithms. Figure 2.20 shows the most popular linear and quadratic convergence schemes. A basic categorization of the linear division algorithms can be done according to the permissible values of each quotient digit generated. In the binary *restoring*, *nonrestoring* or CORDIC algorithms the digits are selected from the set

$$\{0, 1\}.$$

In the binary nonrestoring algorithms a signed-digit set is used, i.e.,

$$\{-1, 1\} = \{\bar{1}, 1\}.$$

In the binary SRT algorithm, named after Sweeney, Robertson, and Tocher [33] who discovered the algorithms at about the same time, the digits from the ternary set

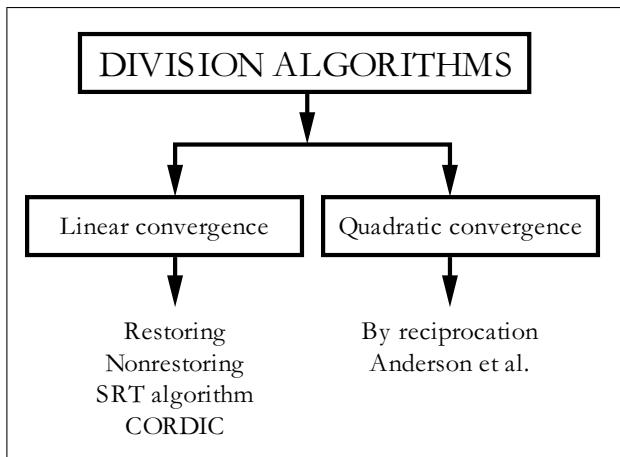


Fig. 2.20. Survey of division algorithms

$$\{-1, 0, 1\} = \{\bar{1}, 0, 1\}$$

are used. All of the above algorithms can be extended to higher radix algorithms. The generalized SRT division algorithms of radix r , for instance, uses the digit set

$$\{-2^r - 1, \dots, -1, 0, 1, \dots, 2^r - 1\}.$$

We find two algorithms with quadratic convergence to be popular. The first algorithm is the division by reciprocation of the denominator, where we compute the reciprocal with the Newton algorithm for finding zeros. The second quadratic convergence algorithms was developed for the IBM 360/91 in the 1960s by Anderson et al. [62]. This algorithm multiplies numerator and denominator with the same factors and converges $N \rightarrow 1$, which results in $D \rightarrow Q$. Note, that the division algorithms with quadratic convergence produce no remainder.

Although the number of iterations in the quadratic convergence algorithms are in the order of $\log_2(b)$ for b bit operands, we must take into account that each iteration step is more complicated (i.e., uses two multiplications) than the linear convergence algorithms, and speed and size performance comparisons have to be done carefully.

2.5.1 Linear Convergence Division Algorithms

The most obvious sequential algorithms is our “pencil-and-paper” method (which we have used many times before) translated into binary arithmetic. We align first the denominator and load the numerator in the remainder register. We then subtract the aligned denominator from the remainder and store the result in the remainder register. If the new remainder is positive

we set the quotient's LSB to 1, otherwise the quotient's LSB is set to zero and we need to restore the previous remainder value by adding the denominator. Finally, we have to realign the quotient and denominator for the next step. The recalculation of the previous remainder is why we call such an algorithm “restoring division.” The following example demonstrates a FSM implementation of the algorithm.

Example 2.18: 8-bit Restoring Divider

The VHDL description⁶ of an 8-bit divider is developed below. Division is performed in four stages. After `reset`, the 8-bit numerator is “loaded” in the remainder register, the 6-bit denominator is loaded and aligned (by 2^{N-1} for a N bit numerator), and the quotient register is set to zero. In the second and third stages, `sub` and `restore`, the actual serial division takes place. In the fourth step, `done`, quotient and remainder are transferred to the output registers. Nominator and quotient are assumed to be 8 bits wide, while denominator and remainder are 6-bit values.

```
-- Restoring Division
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
PACKAGE n_bits_int IS                         -- User defined types
    SUBTYPE SLVN IS STD_LOGIC_VECTOR(7 DOWNTO 0);
    SUBTYPE SLVD IS STD_LOGIC_VECTOR(5 DOWNTO 0);
END n_bits_int;
LIBRARY work; USE work.n_bits_int.ALL;

LIBRARY ieee;                               -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
-----
ENTITY div_res IS                         -----> Interface
    GENERIC(WN : INTEGER := 8;
            WD : INTEGER := 6;
            PO2WND : INTEGER := 8192; -- 2**(WN+WD)
            PO2WN1 : INTEGER := 128; -- 2**((WN-1)
            PO2WN : INTEGER := 255); -- 2**WN-1
    PORT(clk : IN STD_LOGIC;      -- System clock
          reset : IN STD_LOGIC;     -- Asynchronous reset
          n_in : IN SLVN;         -- Nominator
          d_in : IN SLVD;         -- Denumerator
          r_out : OUT SLVD;        -- Remainder
          q_out : OUT SLVN);       -- Quotient
END div_res;
-----
ARCHITECTURE fpga OF div_res IS
    SUBTYPE S14 IS INTEGER RANGE -PO2WND TO PO2WND-1;
    SUBTYPE U8 IS INTEGER RANGE 0 TO PO2WN;
    SUBTYPE U4 IS INTEGER RANGE 0 TO WN;
    TYPE STATE_TYPE IS (ini, sub, restore, done);
```

⁶ The equivalent Verilog code `div_res.v` for this example can be found in Appendix A on page 801. Synthesis results are shown in Appendix B on page 881.

```

SIGNAL state : STATE_TYPE;

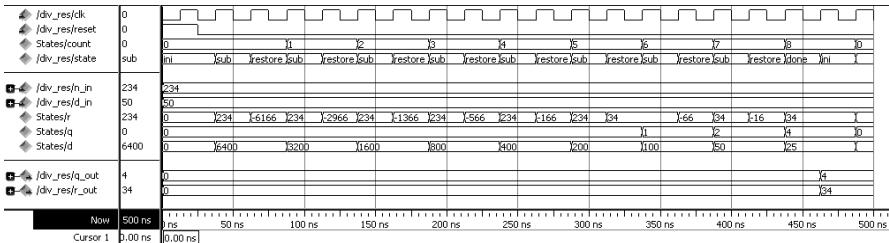
BEGIN
-- Bit width:  WN          WD          WN          WD
--           Nominator / Denumerator = Quotient and Remainder
-- OR:        Nominator = Quotient * Denumerator + Remainder

States: PROCESS(reset, clk)-- Divider in behavioral style
VARIABLE r, d : S14 :=0; -- N+D bit width
VARIABLE q : U8;
VARIABLE count : U4;
BEGIN
IF reset = '1' THEN                      -- asynchronous reset
    state <= ini; q_out <= (OTHERS => '0');
    r_out <= (OTHERS => '0');
ELSIF rising_edge(clk) THEN
CASE state IS
    WHEN ini =>           -- Initialization step
        state <= sub;
        count := 0;
        q := 0;             -- Reset quotient register
        d := PO2WN1 * CONV_INTEGER(d_in); -- Load denumer.
        r := CONV_INTEGER(n_in); -- Remainder = nominator
    WHEN sub =>            -- Processing step
        r := r - d;         -- Subtract denumerator
        state <= restore;
    WHEN restore =>        -- Restoring step
        IF r < 0 THEN
            r := r + d;     -- Restore previous remainder
            q := q * 2;      -- LSB = 0 and SLL
        ELSE
            q := 2 * q + 1; -- LSB = 1 and SLL
        END IF;
        count := count + 1;
        d := d / 2;
        IF count = WN THEN -- Division ready ?
            state <= done;
        ELSE
            state <= sub;
        END IF;
    WHEN done =>           -- Output of result
        q_out <= CONV_STD_LOGIC_VECTOR(q, WN);
        r_out <= CONV_STD_LOGIC_VECTOR(r, WD);
        state <= ini;       -- Start next division
END CASE;
END IF;
END PROCESS States;

END fpga;

```

Figure 2.21 shows the simulation result of a division of 234 by 50. The register **d** shows the aligned denominator values $50 \times 2^7 = 6400, 50 \times 2^6 = 3200, \dots$. Every time the remainder **r** calculated in step **sub** is negative, the previous remainder is restored in step **restore**. In state **done** the quotient 4 and the

**Fig. 2.21.** Simulation results for a restoring divider

remainder 34 are transferred to the output registers of the divider. The design uses 106 LEs, no embedded multiplier, and runs with a registered performance of $F_{max}=263.5$ MHz using the TimeQuest slow 85C model. 2.18

The main disadvantage of the restoring division is that we need two steps to determine one quotient bit. We can combine the two steps using a *non-performing* divider algorithm, i.e., each time the denominator is larger than the remainder, we do *not* perform the subtraction. In VHDL we would write the new step as:

```
t := r - d;          -- temporary remainder value
IF t >= 0 THEN      -- Nonperforming test
    r := t;           -- Use new denominator
    q := q * 2 + 1;  -- LSB = 1 and SLL
ELSE
    q := q * 2;      -- LSB = 0 and SLL
END IF;
```

The number of steps is reduced by a factor of 2 (not counting initialization and transfers of results), as can be seen from the simulation in Fig. 2.22. Note also from the simulation shown in Fig. 2.22 that the remainder r is never negative in the nonperforming division algorithms. On the downside the worst case delay path is increased when compared with the restoring division and the maximum registered performance is expected to be reduced; see Exercise 2.17 (p. 171). The nonperforming divider has two arithmetic operations and the if condition in the worst case path, while the restoring divider has (see step s2) only the if condition and one arithmetic operation in the worst case path.

A similar approach to the nonperforming algorithm, but that does *not* increase the critical path, is the so-called *nonrestoring* division. The idea behind the nonrestoring division is that if we have computed in the restoring division a negative remainder, i.e., $r_{k+1} = r_k - d_k$, then in the next step we will restore r_k by adding d_k and then perform a subtraction of the next aligned denominator $d_{k+1} = d_k/2$. So, instead of adding d_k followed by subtracting $d_k/2$, we can just skip the restoring step and proceed with adding $d_k/2$, when

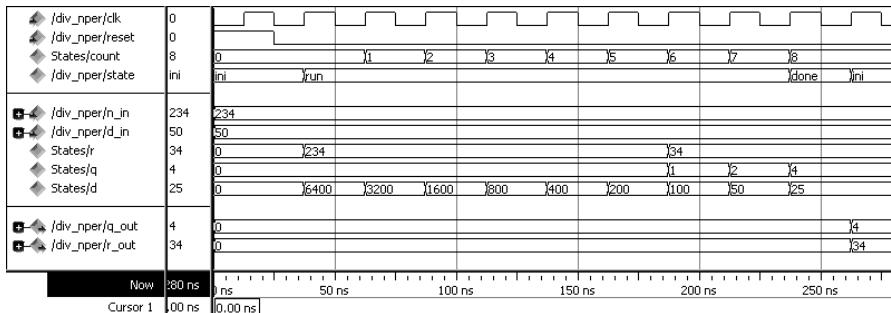


Fig. 2.22. Simulation results for a nonperforming divider

the remainder has (temporarily) a negative value. As a result, we have now quotient bits that can be positive or negative, i.e., $q_k = \pm 1$, but not zero. We can change this signed-digit representation later to a two's complement representation. In conclusion, the nonrestoring algorithms works as follows: every time the remainder after the iteration is positive we store a 1 and subtract the aligned denominator, while for negative remainder, we store a $-1 = \bar{1}$ in the quotient register and add the aligned denominator. To use only one bit in the quotient register we will use a zero in the quotient register to code the -1 . To convert this signed-digit quotient back to a two's complement word, the straightforward way is to put all 1s in one word and the zeros, which are actually the coded $-1 = \bar{1}$ in the second word as a one. Then we need just to subtract the two words to compute the two's complement. On the other hand this subtraction of the -1 s is nothing other than the complement of the quotient augmented by 1. In conclusion, if q holds the signed-digit representation, we can compute the two's complement via

$$q_{2C} = 2 \times q_{SD} + 1. \quad (2.39)$$

Both quotient and remainder are now in the two's complement representation and have a valid result according to (2.36). If we wish to constrain our results in a way that both have the same sign, we need to correct the negative remainder, i.e., for $r < 0$ we correct this via

$$r := r + D \quad \text{and} \quad q := q - 1.$$

Such a nonrestoring divider will now run faster than the nonperforming divider, with about the same registered performance as the restoring divider; see Exercise 2.18 (p. 171). Figure 2.23 shows a simulation of the nonrestoring divider. We notice from the simulation that register values of the remainder are allowed now again to be negative. Note also that the above-mentioned correction for negative remainder is necessary for this value. The not corrected result is $q = 5$ and $r = -16$. The equal sign correction results in $q = 5 - 1 = 4$ and $r = -16 + 50 = 34$, as shown in Fig. 2.23.

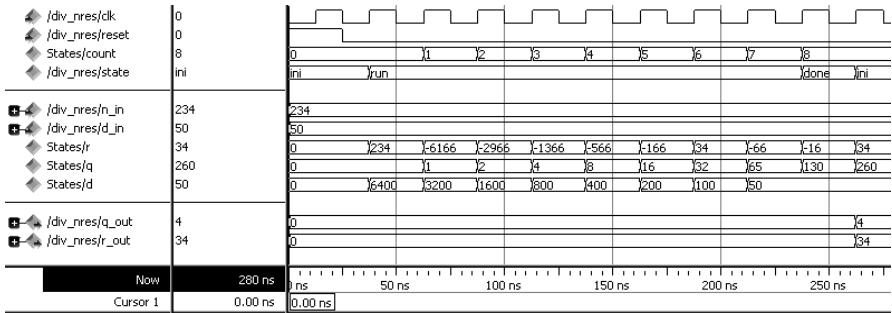


Fig. 2.23. Simulation results for a nonrestoring divider

To shorten further the number of clock cycles needed for the division higher radix (array) divider can be built using, for instance, the SRT and radix 4 coding. This is popular in ASIC designs when combined with the carry-save-adder principle as used in the floating-point accelerators of the Pentium microprocessors. For FPGAs with a limited LUT size this higher-order schemes seem to be less attractive.

A totally different approach to improve the latency are the division algorithms with quadratic convergence, which use fast array multiplier. The two most popular versions of this quadratic convergence schemes are discussed in the next section.

2.5.2 Fast Divider Design

The first fast divider algorithm we wish to discuss is the division through multiplication with the reciprocal of the denominator D . The reciprocal can, for instance, be computed via a look-up table for small bit width. The general technique for constructing iterative algorithms, however, makes use of the Newton method for finding a zero. According to this method, we define a function

$$f(x) = \frac{1}{x} - D \rightarrow 0. \quad (2.40)$$

If we define an algorithm such that $f(x_\infty) = 0$ then it follows that

$$\frac{1}{x_\infty} - D = 0 \quad \text{or} \quad x_\infty = \frac{1}{D}. \quad (2.41)$$

Using the tangent the estimation for the next x_{k+1} is calculated using

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad (2.42)$$

with $f(x) = 1/x - D$ we have $f'(x) = 1/x^2$ and the iteration equation becomes

$$x_{k+1} = x_k - \frac{\frac{1}{x_k} - D}{\frac{-1}{x_k^2}} = x_k(2 - D \times x_k). \quad (2.43)$$

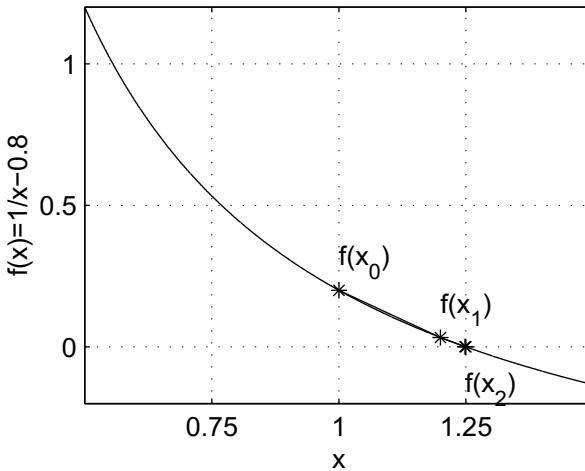


Fig. 2.24. Newton's zero-finding algorithms for $x_\infty = 1/0.8 = 1.25$

Although the algorithm will converge for any initial D , it converges much faster if we start with a normalized value close to 1.0, i.e., we normalized D in such a way that $0.5 \leq D < 1$ or $1 \leq D < 2$ as used for floating-point mantissa, see Sect. 2.7 (p. 109). We can then use an initial value $x_0 = 1$ to get fast convergence. Let us illustrate the Newton algorithm with a short example.

Example 2.19: Newton Algorithm

Let us try to compute the Newton algorithm for $1/D = 1/0.8 = 1.25$. The following table shows in the first column the number of the iteration, in the second column the approximation to $1/D$, in the third column the error $x_k - x_\infty$, and in the last column the equivalent bit precision of our approximation.

k	x_k	$x_k - x_\infty$	Eff. bits
0	1.0	-0.25	2
1	1.2	-0.05	4.3
2	1.248	-0.002	8.9
3	1.25	-3.2×10^{-6}	18.2
4	1.25	-8.2×10^{-12}	36.8

Figure 2.24 shows a graphical interpretation of the Newton zero-finding algorithm. The $f(x_k)$ converges rapidly to zero. 2.19

Because the first iterations in the Newton algorithm only produce a few bits of precision, it may be useful to use a small look-up table to skip the first iterations. A table to skip the first two iterations can, for instance, be found in [33, p. 260].

We note also from the above example the overall rapid convergence of the algorithm. Only 5 steps are necessary to have over 32-bit precision. Many more steps would be required to reach the same precision with the linear convergence algorithms. This quadratic convergence applies for all values not only for our special example. This can be shown as follows:

$$\begin{aligned} e_{k+1} &= x_{k+1} - x_\infty = x_k(2 - D \times x_k) - \frac{1}{D} \\ &= -D \left(x_k - \frac{1}{D} \right)^2 = -De_k^2, \end{aligned}$$

i.e., the error improves in a quadratic fashion from one iteration to the next. With each iteration we double the effective number of bit precision.

Although the Newton algorithm has been successfully used in microprocessor design (e.g., IBM RISC 6000), it has two main disadvantages: First, the two multiplications in each iteration are sequential, and second, the quantization error of the multiplication is accumulated due to the sequential nature of the multiplication. Additional guard bits are used in general to avoid this quantization error.

The following convergence algorithm, although similar to the Newton algorithm, has an improved quantization behavior and uses 2 multiplications in each iteration that can be computed parallel. In the *convergence division* scheme both numerator N and denominator D are multiplied by approximation factors f_k , which, for a sufficient number of iterations k , we find

$$D \prod f_k \rightarrow 1 \quad \text{and} \quad N \prod f_k \rightarrow Q. \quad (2.44)$$

This algorithm, originally developed for the IBM 360/91, is credited to Anderson et al. [62], and the algorithm works as follows:

Algorithm 2.20: Division by Convergence

- 1) Normalize N and D such that D is close to 1. Use a normalization interval such as $0.5 \leq D < 1$ or $1 \leq D < 2$ as used for floating-point mantissa.
- 2) Initialize $x_0 = N$ and $t_0 = D$.
- 3) Repeat the following loop until x_k shows the desired precision.

$$f_k = 2 - t_k$$

$$x_{k+1} = x_k \times f_k$$

$$t_{k+1} = t_k \times f_k$$

It is important to note that the algorithm is self-correcting. Any quantization error in the factors does not really matter because numerator and denominator are multiplied with the same factor f_k . This fact has been used in the IBM 360/91 design to reduce the required resources. The multiplier used for the first iteration has only a few significant bits, while in later iteration more multiplier bits are allocated as the factor f_k gets closer to 1.

Let us demonstrate the multiply by convergence algorithm with the following example.

Example 2.21: Anderson–Earle–Goldschmidt–Powers Algorithm

Let us try to compute the division-by-convergence algorithm for $N = 1.5$ and $D = 1.2$, i.e., $Q = N/D = 1.25$. The following table shows in the first column the number of the iteration, in the second column the scaling factor f_k , in the third column the approximation to N/D , in the fourth column the error $x_k - x_\infty$, and in the last column the equivalent bit precision of our approximation.

k	f_k	x_k	$x_k - x_\infty$	Eff. bits
0	$0.8 \approx \frac{205}{256}$	$1.5 \approx \frac{384}{256}$	0.25	2
1	$1.04 \approx \frac{267}{256}$	$1.2 \approx \frac{307}{256}$	-0.05	4.3
2	$1.0016 \approx \frac{257}{256}$	$1.248 \approx \frac{320}{256}$	0.002	8.9
3	$1.0 + 2.56 \times 10^{-6}$	1.25	-3.2×10^{-6}	18.2
4	$1.0 + 6.55 \times 10^{-12}$	1.25	-8.2×10^{-12}	36.8

We notice the same quadratic convergence as in the Newton algorithm; see Example 2.19 (p. 101).

The VHDL description⁷ of an 8-bit fast divider is developed below. We assume that denominator and numerator are normalized as, for instance, typical for floating-point mantissa values, to the interval $1 \leq N, D < 2$. This normalization step may require essential addition resources (leading-zero detection and two barrelshifters) when the denominator and numerator are not normalized. Nominator, denominator, and quotient are all assumed to be 9 bits wide. The decimal values 1.5, 1.2, and 1.25 are represented in a 1.8-bit format (1 integer and 8 fractional bits) as $1.5 \times 256 = 384$, $1.2 \times 256 = 307$, and $1.25 \times 256 = 320$, respectively. Division is performed in three stages. First, the 1.8-formatted denominator and numerator are loaded into the registers. In the second state, `run`, the actual convergence division takes place. In the third step, `done`, the quotient is transferred to the output register.

```
-- Convergence division after Anderson, Earle, Goldschmidt,
LIBRARY ieee; USE ieee.std_logic_1164.ALL; -- and Powers

PACKAGE n_bits_int IS          -- User defined types
  SUBTYPE U3 IS INTEGER RANGE 0 TO 7;
  SUBTYPE U10 IS INTEGER RANGE 0 TO 1023;
  SUBTYPE SLVN IS STD_LOGIC_VECTOR(8 DOWNTO 0);
  SUBTYPE SLVD IS STD_LOGIC_VECTOR(8 DOWNTO 0);
END n_bits_int;

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
-----
```

⁷ The equivalent Verilog code `div_aegp.v` for this example can be found in Appendix A on page 803. Synthesis results are shown in Appendix B on page 881.

```

ENTITY div_aegp IS                               -----> Interface
  GENERIC(WN : INTEGER := 9; -- 8 bit plus one integer bit
          WD : INTEGER := 9;
          STEPS : INTEGER := 2;
          TWO : INTEGER := 512;           -- 2**((WN+1)
          PO2WN : INTEGER := 256;       -- 2**((WN-1)
          PO2WN2 : INTEGER := 1023);    -- 2**((WN+1)-1
  PORT (clk    : IN STD_LOGIC;      -- System clock
        reset : IN STD_LOGIC;      -- Asynchronous reset
        n_in  : IN SLVN;          -- Nominator
        d_in  : IN SLVD;          -- Denumerator
        q_out : OUT SLVD);       -- Quotient
END div_aegp;
-- -----
ARCHITECTURE fpga OF div_aegp IS

  TYPE STATE_TYPE IS (ini, run, done);
  SIGNAL state     : STATE_TYPE;

BEGIN
  -- Bit width: WN          WD          WN          WD
  --             Nominator / Denumerator = Quotient and Remainder
  -- OR:         Nominator = Quotient * Denumerator + Remainder

  States: PROCESS(reset, clk)-- Divider in behavioral style
    VARIABLE x, t, f : U10 := 0; -- WN+1 bits
    VARIABLE count : INTEGER RANGE 0 TO STEPS;
  BEGIN
    IF reset = '1' THEN                  -- Asynchronous reset
      state <= ini;
      q_out <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
      CASE state IS
        WHEN ini =>                   -- Initialization step
          state <= run;
          count := 0;
          t := CONV_INTEGER(d_in); -- Load denominator
          x := CONV_INTEGER(n_in); -- Load nominator
        WHEN run =>                  -- Processing step
          f := TWO - t;
          x := x * f / PO2WN;
          t := t * f / PO2WN;
          count := count + 1;
        IF count = STEPS THEN -- Division ready ?
          state <= done;
        ELSE
          state <= run;
        END IF;
        WHEN done =>                -- Output of results
          q_out <= CONV_STD_LOGIC_VECTOR(x, WN);
          state <= ini;            -- start next division
      END CASE;
    END IF;
  END;

```

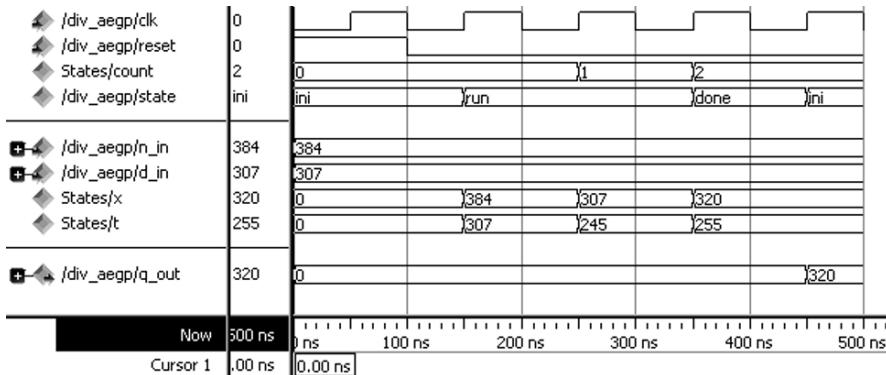


Fig. 2.25. Simulation results for a convergence divider

```
END PROCESS States;
```

```
END fpga;
```

Figure 2.25 shows the simulation result of the division $1.5/1.2$. The variable **f** (which becomes an internal net and is not shown in the simulation) holds the three scaling factors 205, 267, and 257, sufficient for 8-bit precision results. The **x** and **t** values are multiplied by the scaling factor **f** and scaled down to the 1.8 format. **x** converges to the quotient $1.25=320/256$, while **t** converges to $1.0=255/256$, as expected. In state **done** the quotient $1.25 = 320/256$ is transferred to the output registers of the divider. Note that the divider produces no remainder. The design uses 45 LEs, four embedded multipliers and runs with a registered performance of **Fmax**=124.91 MHz using the **TimeQuest** slow 85C model.

2.21

Although the registered performance of the nonrestoring divider (see Fig. 2.23) is about twice as high, the total latency, however, in the convergence divider is reduced, because the number of processing steps are reduced from 8 to $\lceil \sqrt{8} \rceil = 3$ (not counting initialization in both algorithms). The convergence divider uses less LEs as the nonrestoring divider but also four embedded multipliers.

2.5.3 Array Divider

Obviously, as with multipliers, all division algorithms can be implemented in a sequential, FSM-like, way or in the array form. If the array form and pipelining is desired, a good option will then be to use the **lpm_divide** block, which implements an array divider with the option of pipelining. If no pipelining is required then Quartus II 12.1 is able to synthesize behavior code such as

```
SIGNAL n, d, q : INTEGER RANGE -128 TO 127;
```

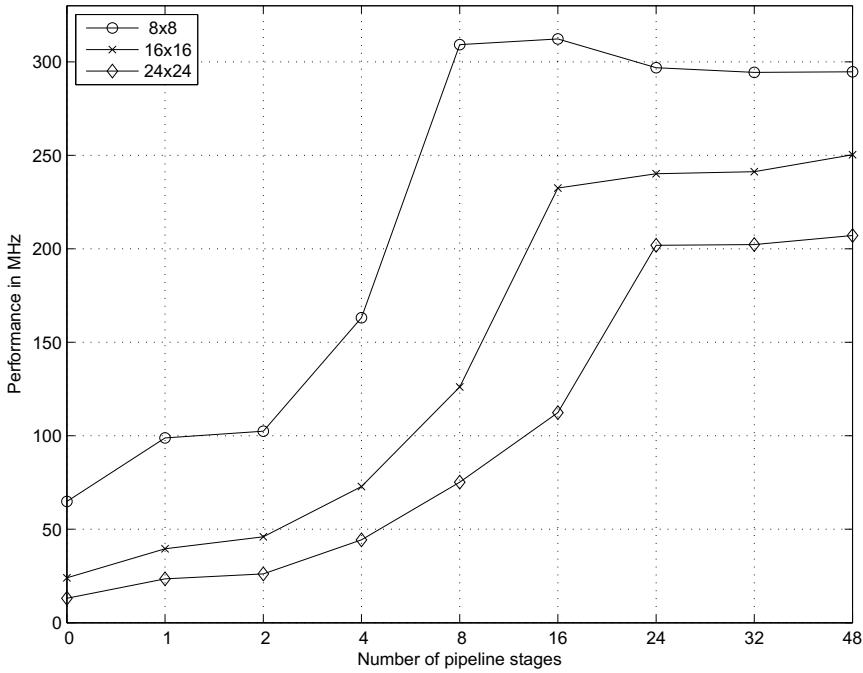


Fig. 2.26. Performance of array divider using the `lpm_divide` macro block. The behavior code corresponds to zero pipeline stages

```
...
q <= n / d;
```

Figure 2.26 shows the registered performance using the `TimeQuest` slow 85C model and Fig. 2.27 the LEs necessary for 8×8 -, 16×16 -, and 24×24 -bit array dividers including the 4 I/O registers. Note the logarithmic like scaling for the number of pipeline stages on the horizontal axis. We conclude from the performance measurement that the optimal number of pipeline stages is the same as the number of bits in the denominator.

2.6 Fixed-Point Arithmetic Implementation

A substantial addition in VHDL-2008 relevant to DSP is the introduction of the unsigned and signed fixed-point data types `ufixed` and `sfixed` and the `float` data types discussed in the next section. Based on the Appendix G (pp. 522–537) of the VHDL-2008 LRM several textbooks [63–65] now cover this new data types and the operations too. Since in DSP we more often deal with signed than unsigned numbers, let us in the following focus on the `sfixed` data types. Although it is part of the VHDL-2008 standard, efforts

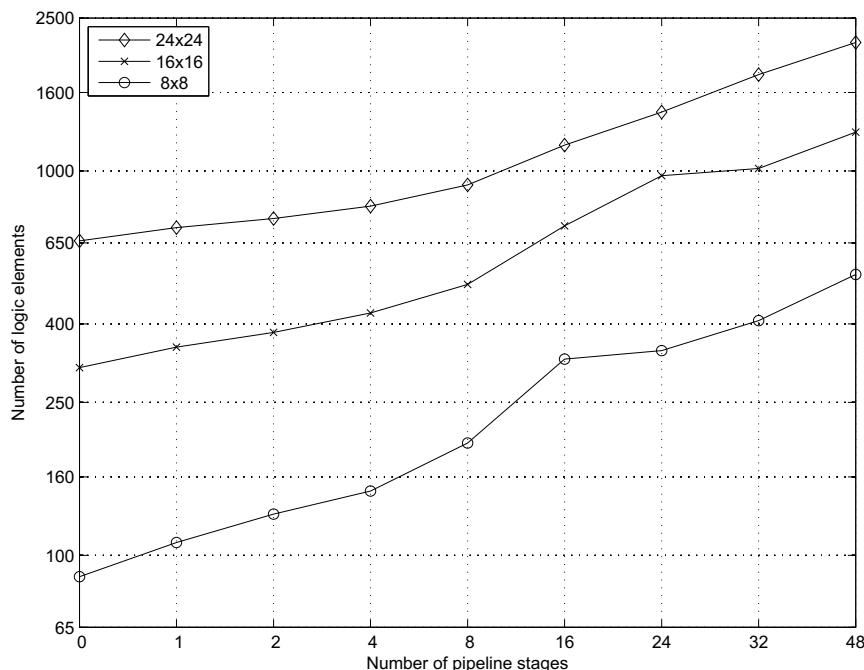


Fig. 2.27. Effort in LEs for array divider using the `lpm_divide` macro block including the 4 I/O registers. The behavior code corresponds to zero pipeline stages

have been made for legacy support that the package is also available for VHDL-1993 through the call of an additional library. There are two files to be included. The first includes the type configuration definitions and the second library file includes the operation and conversion functions. In VHDL-1993 we would write

```
LIBRARY ieee_proposed;
USE ieee_proposed.fixed_float_types.ALL;
USE ieee_proposed.fixed_pkg.all;
```

These files are available free of charge from www.eda.org/fphdl. The library is a complete package of operations and functions written by David Bishop and includes over 8K lines of VHDL code. At the time of writing, an equivalent Verilog code is, according to David Bishop, currently in progress and it is the hope that a Verilog version may become part of a future standard too. Since most vendors support only a subset of standard VHDL language, small modified versions are available that have been tested for Altera, Xilinx, Synopsys, Cadence, and MentorGraphics (ModelSim) tools. Here is a brief listing of supported operations and functions relevant to DSP for the `sfixed` data type:

Arithmetic	<code>+ , - , * , / , ABS , REM , MOD , ADD_CARRY</code>
Logical	<code>NOT , AND , NAND , OR , NOR , XOR , XNOR</code>
Shift	<code>SLL , SRL , ROL , ROR , SLA , SRA</code>
Comparison	<code>= , /= , > , < , >= , <=</code>
Conversion	<code>TO_SLV , TO_SFIXED , TO_UFIXED</code>
Others	<code>RESIZE , SCALB , MAXIMUM , MINIMUM</code>

Some of the arithmetic functions (i.e., divide, reciprocal, remainder, modulo) are available as function calls that allow a specification of additional parameters such as rounding modes, overflow style, and guard bits. As rounding style for the `sfixed` we have to choose between the style `fixed_round` and `fixed_truncate`. For the overflow the two styles are `fixed_saturate` and `fixed_wrap`. The default is `fixed_saturate` and `fixed_round`. Minimum HW resources are used if we use `fixed_wrap`, and `fixed_truncate`, or 0 guard bits. The `sfixed` library is designed to minimize the chance of overflow error. This is a departure from most other data types and as a consequence the result of an addition operation needs to have one addition guard bit to the left, i.e.,

```
...
SIGNAL a : SFIXED(3 DOWNTO -4) := TO_SFIXED(3.625,3,-4);
SIGNAL b : SFIXED(3 DOWNTO -4);
SIGNAL s : SFIXED(4 DOWNTO -4);
SIGNAL r : SFIXED(3 DOWNTO -4);

...
s <= a + b; -- Coding ok; needs 9 LEs
r <= a + b; -- not allowed
--Error:expression has 9 elements, but must have 8 elements
r <= RESIZE(a + b, r); -- Needs 16 LEs
r <= RESIZE(a + b, r, fixed_wrap,fixed_truncate);
-- Needs 8 LEs
```

We see that the operands `a` and `b` have 4-bit integer (index: 3...0) and 4 fractional bits (index: -1...-4). The value $a = 3.625$ would be represented in the `sfixed` format as follows:

<code>a =</code>	Bit no.:	3	2	1	0	-1	-2	-3	-4
	Weight	8	4	2	1	0.5	0.25	0.125	0.0625
	Coding	0	0	1	1	1	0	1	0

Note how convenient it is to convert the real numbers into `sfixed` format using `TO_SFIXED()`. This make a code with different coefficients much easier to read.

However, as mentioned, the `sfixed` format is strict in the requirement of additional guard bits such that the above statement of `s <= a + b;` will require nine LEs. The statement `r <= a + b` would not be valid, since `r` is of the same size as `a` and `b`. One additional integer bit is used in the result.

If you want the output in a 4.4 format like the inputs then you can use the `resize` method

```
r <= RESIZE(a+b, r);
```

where the second parameter specifies the left and right bounds of the result. The synthesis result now comes out even larger due to the fact that by default, the library uses `fixed_saturate` and `fixed_round` options. Three multiplexer are implemented and a total of 16 LEs are needed. To avoid the `saturate` and `rounding` we would code

```
r <= RESIZE(a + b, r, fixed_wrap, fixed_truncate);
```

and the synthesis result will be the desired eight LEs. Let us now assume the two numbers have range format $A_L \dots A_R$ and $B_L \dots B_R$ format; then Table 2.9 shows the bit width requirements for the popular `sfixed` operations.

Table 2.9. Index range of typical `sfixed` operation

Operations	Result range	Same range: $A_L=B_L=L$; $A_R=B_R=R$
$A+B$, $A-B$	$\max(A_L, B_L)+1 \dots \min(A_R, B_R)$	$L+1 \dots R$
$\text{abs}(A), -A$	$A_L+1 \dots A_R$	$L+1 \dots R$
$A*B$	$A_L+B_L+1 \dots A_R+B_R$	$2L+1 \dots 2R$
A/B	$A_L-B_R+1 \dots A_R-B_L$	$L-R+1 \dots R-L$
$\text{reciprocal}(A)$	$-A_R \dots -A_L-1$	$-R \dots -L-1$

2.7 Floating-Point Arithmetic Implementation

Due to the large gate count capacity of current FPGAs the design of floating-point arithmetic has become a viable option. In addition, the introduction of the embedded 18×18 -bit array multiplier in Altera Stratix or Cyclone and Xilinx Virtex or Spartan FPGA device families allows an efficient design of custom floating-point arithmetic. We will therefore discuss the design of basic building blocks such as a floating-point adder, subtractor, multiplier, reciprocals and divider, and the necessary conversion blocks to and from fixed-point data format. Such blocks are available from several IP providers, and are now part of the 1076-2008 IEEE VHDL standard. Details can be found in Appendix G of the LRM. Legacy support to 1076-1993 VHDL is possible through an additional fixed- and floating-point package that we will discuss later.

Most of the commercially available floating-point blocks use five or more pipeline stages to increase the throughput. To keep the presentation simple we

will not use pipelining and indeed the VHDL1076-2008 floating-point library does not use any pipelining. Later we will design a 32-bit floating-point unit, but as initial custom floating-point format we will use the (1,6,5) floating-point format introduced in Sect. 2.2.3 (p. 75). This format uses one sign bit, six bits for the exponent and five bits for the mantissa. We support special coding for zero and infinities; support for NaNs and denormals are also enabled as standard in the IEEE 1076-2008 package. Rounding is done via round-to-nearest. By default three guard bits are added to the bottom of every operation. The fixed-point format used in the examples has six integer bits (including a sign bit) and six fractional bits.

2.7.1 Fixed-Point to Floating-Point Format Conversion

As shown in Sect. 2.2.3 (p. 75), floating-point numbers use a signed-magnitude format and the first step is therefore to convert the two's complement number to signed-magnitude form. If the sign of the fixed-point number is one, we need to compute the complement of the fixed-point number, which becomes the unnormalized mantissa. In the next step we normalize the mantissa and compute the exponent. For the normalization we first determine the number of leading zeros. This can be done with a **LOOP** statement within a sequential **PROCESS** in VHDL. Using this number of leading zeros, we shift the mantissa left, until the first 1 “leaves” the mantissa registers, i.e., the hidden one is also removed. This shift operation is actually the task of a barrelshifter, which can be inferred in VHDL via the **SLL** instruction. Unfortunately the **SLL** was only defined for **BIT_VECTOR** data type in the 1076-1993 standard, but not for the **STD_LOGIC_VECTOR** data type. The newer VHDL 1076-2008 compiler also supports the **SLL** for **STD_LOGIC_VECTOR**. However, we can design a barrelshifter in many different ways as Exercise 2.19 (p. 172) shows. Another alternative would be to design a function overloading for the **STD_LOGIC_VECTOR** that allows a shift operation; see Exercise 1.20, p. 53.

The exponent of our floating-point number is computed as the sum of the bias and the number of integer bits in our fixed-point format minus the leading zeros in the unnormalized mantissa.

Finally, we concatenate the sign, exponent, and the normalized mantissa to a single floating-point word if the fixed-point number is not zero, otherwise we also set the floating-point word to zero.

We have assumed that the range of the floating-point number is larger than the range of the fixed-point number, i.e., the special number ∞ will never be used in the conversion.

Figure 2.28 shows the conversion from 12-bit fixed-point data to the (1,6,5) floating-point data for five values ± 1 , absolute maximum, absolute minimum, and the smallest value. Rows one to three show the 12-bit fixed-point number and the integer and fractional parts. Rows four to seven show the complete floating-point number, followed by the three parts, sign, exponent, and mantissa.

op		fix2fp	fix2fp			
Inputs:						
Fix	000001000000	000001000000	111111000000	011111111111	1000000000001	0000000000001
int	000001	000001	111111	011111	100000	000000
Frac	000000	000000		111111	100001	
Output:						
Fp	001111100000	001111100000	101111100000	010010000000	110010000000	001100100000
sign	0					
exp	011111	011111		100100		011001
man	00000	00000				

Fig. 2.28. Simulation results for a (1,5,6) fixed-point format to (1,6,5) floating-point conversion. The five represented values are: +1; -1; maximum \approx 32; minimum = -32; and smallest = 1/64

2.7.2 Floating-Point to Fixed-Point Format Conversion

The floating-point to fixed-point conversion is, in general, more complicated than the conversion in the other direction. Depending on whether the exponent is larger or smaller than the bias we need to implement a left or right shift of the mantissa. In addition, extra consideration is necessary for the special values $\pm\infty$ and ± 0 .

To keep the discussion as simple as possible, we assume in the following that the floating-point number has a larger dynamic range than the fixed-point number, but the fixed-point number has a higher precision, i.e., the number of fractional bits of the fixed-point number is larger than the bits used for the mantissa in the floating-point number.

The first step in the conversion is the correction of the bias in the exponent. We then place the hidden 1 to the left and the (fractional) mantissa to the right of the decimal point of the fixed-point word. We then check whether the exponent is too large to be represented with the fixed-point number and then set the fixed-point number to the maximum value. Also, if the exponent is too small, we set the output value to zero. If the exponent is in the valid range where the floating-point number can be represented with the fixed-point format, we shift left the 1.m mantissa value (format see (2.23), p. 75) for positive exponents, and shift right for negative exponent values. This, in general, can be coded with the SLL and SRL in VHDL, respectively, but these `BIT_VECTOR` operations are not supported in VHDL-1993 for `STD_LOGIC_VECTOR`; see Exercise 1.20, p. 53. In the final step we convert the signed magnitude representation to the two's complement format by evaluating the sign bit of the floating-point number.

Figure 2.29 shows the conversion from (1,6,5) floating-point format to (1,5,6) fixed-point data for the five values ± 1 , absolute maximum, absolute minimum, and the smallest value. Rows one to four show the 12-bit floating-point number and the three parts, sign, exponent, and mantissa. Rows five to seven show the whole fixed-point number, followed by the integer and fractional parts. Note that the conversion is without any quantization error for ± 1 and the smallest value. For the absolute maximum and minimum

op	fp2fix				
Inputs:					
fp	001111100000	001111100000	101111100000	010010000000	110010000000
sign	0				
exp	011111	011111		100100	011001
man	00000	00000			
Output:					
fix	000001000000	000001000000	111111000000	011111111111	1000000000000
int	000001	000001	111111	011111	100000
frac	000000	000000		111111	000000

Fig. 2.29. Simulation results for (1,6,5) floating-point format to (1,5,6) fixed-point format conversion. The represented values are: +1; -1; max \approx 32; min = -32; and smallest = 1/64

values, however, the smaller precision in the floating-point numbers gives the imperfect conversion values compared with Fig. 2.28.

2.7.3 Floating-Point Multiplication

In contrast to fixed-point operations, multiplication in floating-point is the simplest of all arithmetic operations and we will discuss this first. In general, the multiplication of two numbers in scientific format is accomplished by multiplication of the mantissas and adding of the exponents, i.e.,

$$f_1 \times f_2 = (a_1 2^{e_1}) \times (a_2 2^{e_2}) = (a_1 \times a_2) 2^{e_1 + e_2}.$$

For our floating-point format with an implicit one and a biased exponent this becomes

$$\begin{aligned} f_1 \times f_2 &= (-1)^{s_1} (1.m_1 2^{e_1 - \text{bias}}) \times (-1)^{s_2} (1.m_2 2^{e_2 - \text{bias}}) \\ &= (-1)^{s_1 + s_2 \bmod 2} \underbrace{(1.m_1 \times 1.m_2)}_{m_3} 2^{\underbrace{e_1 + e_2 - \text{bias}}_{e_3} - \text{bias}} \\ &= (-1)^{s_3} 1.m_3 2^{e_3 - \text{bias}}. \end{aligned}$$

We note that the exponent sum needs to be adjusted by the bias, since the bias is included twice in both exponents. The sign of the product is the XOR or modulo-2 sum of the two sign bits of the two operands. We also need to take care of the special values. If one factor is ∞ the product should be ∞ too. Next, we check whether one factor is zero and set the product to zero if true. Because we do not support NaNs, this implies that $0 \times \infty$ is set to ∞ . Special values may also be produced from original nonspecial operands. If we detect an overflow, i.e.,

$$e_1 + e_2 - \text{bias} \geq E_{\max},$$

we set the product to ∞ . Likewise, if we detect an underflow, i.e.,

$$e_1 + e_2 - \text{bias} \leq E_{\min},$$

op	mul					
<hr/>						
<hr/>						
Inputs:						
1. FP number:						
a	011...	101111100000	001111110000	00000111100000	01111110000000	01111110000000
sign	0					
exp	111111	011111		000111	111110	111111
man	00000	00000	11000	10000	00000	
2. FP number:						
b	000...	101111100000	001111110000	00000111100000	01111110000000	01111110000000
sign	0					
exp	000000	011111		000111	111110	111111
man	00000	00000	11000	10000	00000	
Output:						
r	011...	001111100000	010000010001	00000000000000	011111100000	011111100001
sign	0					
exp	111111	011111	100000	000000	111111	
man	00001	00000	10001	00000		00001

Fig. 2.30. Simulation results for multiplications with floating-point numbers in the (1,6,5) format

we set the product to zero. It can be seen that the internal representation of the exponent e_3 of the product must have two more bits than the two factors, because we need a sign and a guard bit. Fortunately, the normalization of the product $1.m_3$ is relatively simple; because both operands are in the range $1.0 \leq 1.m_{1,2} < 2.0$, the mantissa product is therefore in the range $1.0 \leq 1.m_3 < 4.0$, i.e., a shift by one bit (and exponent adjustment by 1) is sufficient to normalize the product.

Finally, we build the new floating-point number by concatenation of the sign, exponent, and magnitude.

Figure 2.30 shows the multiplication in the (1,6,5) floating-point format of the following values:

- 1) $(-1) \times (-1) = 1.0_{10} = 1.00000_2 \times 2^{31-\text{bias}}$
- 2) $1.75 \times 1.75 = 3.0625_{10} = 11.0001_2 \times 2^{31-\text{bias}} = 1.10001_2 \times 2^{32-\text{bias}}$
- 3) exponent: $7 + 7 - \text{bias} = -17 < E_{\min} \rightarrow$ underflow in multiplication
- 4) exponent: $62 + 62 - \text{bias} = 93 \geq E_{\max} \rightarrow$ overflow
- 5) $\infty \times \infty = \infty$
- 6) $0 \times \infty = \text{NaN}$

Rows one to four show the first floating-point number **a** and the three parts: sign, exponent, and mantissa. Rows five to eight show the same for the second operand **b**, and rows nine to 12 the product **r** and the decomposition of the three parts.

2.7.4 Floating-Point Addition

Floating-point addition is more complex than multiplication. Two numbers in scientific format

$$f_3 = f_1 + f_2 = (a_1 2^{e_1}) \pm (a_2 2^{e_2})$$

can only be added if the exponents are the same, i.e., $e_1 = e_2$. Without loss of generality we assume in the following that the second number has the (absolute) smaller value. If this is not true, we just exchange the first and second numbers. The next step is now to “denormalize” the smaller number by using the following identity:

$$a_2 2^{e_2} = a_2 / 2^d 2^{e_2+d}.$$

If we select the normalization factor such as $e_2 + d = e_1$, i.e., $d = e_1 - e_2$, we get

$$a_2 / 2^d 2^{e_2+d} = a_2 / 2^{e_1-e_2} 2^{e_1}.$$

Now both numbers have the same exponent and we can, depending on the signs, add or subtract the first mantissa and the aligned second, according to

$$a_3 = a_1 \pm a_2 / 2^{e_1-e_2}.$$

We also need to check whether the second operand is zero. This is the case if $e_2 = 0$ or $d > M$, i.e., the shift operation reduces the second mantissa to zero. If the second operand is zero the first (larger) operand is forwarded to the result f_3 .

The two aligned mantissas are added if the two floating-point operands have the same sign, otherwise subtracted. The new mantissa needs to be normalized to have the $1.m_3$ format, and the exponent, initially set to $e_3 = e_1$, needs to be adjusted accordingly to the normalization of the mantissa. We need to determine the number of leading zeros including the first one and perform a shift logic left (SLL). We also need to take into account whether one of the operands is a special number, or whether over- or underflow occurs. If the first operand is ∞ or the new computed exponent is larger than E_{\max} the output is set to ∞ . A NaN results for $\infty - \infty$. If the new computed exponent is smaller than E_{\min} , underflow has occurred and the output is set to zero. Finally, we concatenate the sign, exponent, and mantissa to the new floating-point number.

Figure 2.31 shows the addition in the (1,6,5) floating-point format of the following values:

- 1) $1.0 + (-1.0) = 0$
- 2) $9.25 + (-10.5) = -1.25_{10} = -1.01000_2 \times 2^{31-\text{bias}}$
- 3) $1.00111_2 \times 2^{2-\text{bias}} + (-1.00100_2 \times 2^{2-\text{bias}}) = 0.00011_2 \times 2^{2-\text{bias}} = 1.1_2 \times 2^{-2-\text{bias}} \rightarrow -2 < E_{\min} \rightarrow \text{underflow} \rightarrow \text{denormalized number}$
- 4) $1.01111_2 \times 2^{62-\text{bias}} + 1.11110_2 \times 2^{62-\text{bias}} = 11.01101_2 2^{62-\text{bias}} = 1.12^{63-\text{bias}} \rightarrow 63 \geq E_{\max} \rightarrow \text{overflow}$
- 5) $-\infty + 1 = -\infty$
- 6) $\infty - \infty = \text{NaN}$

Rows one to four show the first floating-point number **a** and the three parts: sign, exponent, and mantissa. Rows five to eight show the same for the second operand **b**, and rows nine to 12 show the sum **r** and the decomposition in the three parts, sign, exponent, and mantissa.

op	add						
Inputs:							
1. FP number:							
a	01...	001111100000	010001000101	000001000111	011111001111	111111100000	011111100000
sign	0						
exp	111111	011111	100010	000010	111110	111111	
man	00000	00000	00101	00111	01111	00000	
2. FP number:							
b	11...	101111100000	110001001010	100001000100	011111011110	001111100000	111111100000
sign	1						
exp	111111	011111	100010	000010	111110	011111	111111
man	00000	00000	01010	00100	11110	00000	
Output:							
r	01...	000000000000	101111101000	0000000000110	011111100000	111111100000	011111100001
sign	0						
exp	111111	000000	011111	000000	111111		
man	00001	00000	01000	00110	100000		00001

Fig. 2.31. Simulation results for additions with floating-point numbers in the (1,6,5) format

2.7.5 Floating-Point Division

In general, the division of two numbers in scientific format is accomplished by division of the mantissas and subtraction of the exponents, i.e.,

$$f_1/f_2 = (a_1 2^{e_1}) / (a_2 2^{e_2}) = (a_1/a_2) 2^{e_1 - e_2}.$$

For our floating-point format with an implicit one and a biased exponent this becomes

$$\begin{aligned} f_1/f_2 &= (-1)^{s_1} (1.m_1 2^{e_1 - \text{bias}}) / (-1)^{s_2} (1.m_2 2^{e_2 - \text{bias}}) \\ &\quad \underbrace{e_1 - e_2 - \text{bias} + \text{bias}}_{e_3} \\ &= (-1)^{s_1 + s_2 \bmod 2} \underbrace{(1.m_1 / 1.m_2) 2}_{m_3}^{\quad e_3} \\ &= (-1)^{s_3} 1.m_3 2^{e_3 + \text{bias}}. \end{aligned}$$

We note that the exponent sum needs to be adjusted by the bias, since the bias is no longer present after the subtraction of the exponents. The sign of the division is the XOR or modulo-2 sum of the two sign bits of the two operands. The division of the mantissas can be implemented with any algorithm discussed in Sect. 2.5 (p. 93) or we can use the `lpm_divide` component. Because the denominator and quotient has to be at least $M+1$ bits wide, but numerator and quotient have the same bit width in the `lpm_divide` component, we need to use numerator and quotient with $2 \times (M+1)$ bits. Because the numerator and denominator are both in the range $1 \leq 1.m_{1,2} < 2$, we conclude that the quotient will be in the range $0.5 \leq 1.m_3 < 2$. It follows that a normalization of only one bit (including the exponent adjustment by 1) is required.

We also need to take care of the special values. The result is ∞ if the numerator is ∞ , the denominator is zero, or we detect an overflow, i.e.,

op	div							
<u>Inputs:</u>								
1. FP number:								
a	000...	101111100000	110001001010	010001000101	011110010000	000001100000	001111100000	000000000000
sign	0							
exp	000000	011111	100010		111100	000011	011111	000000
man	00000	00000	01010	00101	10000	00000		
2. FP number:								
b	000...	101111100000	010001000101	110001001010	000001100000	111110010000	000000000000	
sign	0							
exp	000000	011111	100010		000011	111100	000000	
man	00000	00000	00101	01010	00000	10000	00000	
<u>Output:</u>								
r	011...	001111100000	101111100100	101111101100	011111100000	100000000000	011111100000	011111100001
sign	0							
exp	111111	011111		011110	011111	000000	011111	
man	00001	00000	00100	11000	00000			00001

Fig. 2.32. Simulation results for division with floating-point numbers in the (1,6,5) format

$$e_1 - e_2 + \text{bias} = e_3 \geq E_{\max}.$$

Then we check for a zero quotient. The quotient is set to zero if the numerator is zero, denominator is ∞ , or we detect an underflow, i.e.,

$$e_1 - e_2 + \text{bias} = e_3 \leq E_{\min}.$$

In all other cases the result is in the valid range that produces no special result.

Finally, we build the new floating-point number by concatenation of the sign, exponent, and magnitude.

Figure 2.32 shows the division in the (1,6,5) floating-point format of the following values:

- 1) $(-1)/(-1) = 1.0_{10} = 1.00000_2 \times 2^{31-\text{bias}}$
- 2) $-10.5/9.25_{10} = -1.\overline{135}_{10} \approx -1.001_2 \times 2^{31-\text{bias}}$
- 3) $9.25/(-10.5)_{10} = -0.880952_{10} \approx -1.11_2 \times 2^{30-\text{bias}}$
- 4) exponent: $60 - 3 + \text{bias} = 88 > E_{\max} \rightarrow$ overflow in division
- 5) exponent: $3 - 60 + \text{bias} = -26 < E_{\min} \rightarrow$ underflow in division
- 6) $1.0/0 = \infty$
- 7) $0/0 = \text{NaN}$

Rows one to four show the first floating-point number and the three parts: sign, exponent, and mantissa. Rows five to eight show the same for the second operand, and rows nine to 12 show the quotient and the decomposition in the three parts.

2.7.6 Floating-Point Reciprocal

Although the reciprocal function of a floating-point number, i.e.,

$$\begin{aligned} 1.0/f &= \frac{1.0}{(-1)^s 1.m 2^e} \\ &= (-1)^s 2^{-e} / 1.m \end{aligned}$$

op	rec	rec			
Inputs:					
+ a	011...	1100000000000	101111101000	101111100001	0000000000000
sign	0				
+ exp	111111	100000	011111		000000
+ man	00000	00000	01000	00001	00000
Output:					
+ r	000...	101111000000	101111010011	101111011110	011111100000
sign	0				
+ exp	000000	011110		111111	000000
+ man	00000	00000	10011	11110	00000

Fig. 2.33. Simulation results for reciprocal with floating-point numbers in the (1,6,5) format

seems to be less frequently used than the other arithmetic functions, it is nonetheless useful since it can also be used in combination with the multiplier to build a floating-point divider, because

$$f_1/f_2 = \frac{1.0}{f_2} \times f_1,$$

i.e., the reciprocal of the denominator followed by multiplication is equivalent to the division.

If the bit width of the mantissa is not too large, we may implement the reciprocal of the mantissa via a look-up table implemented with a `case` statement or with a M9K memory block [34]. Because the mantissa is in the range $1 \leq 1.m < 2$, the reciprocal must be in the range $0.5 < \frac{1}{1.m} \leq 1$. The mantissa normalization is therefore a 1-bit shift for all values except $f = 1.0$.

We also need to take care of the special values. The reciprocal of ∞ is 0, and the reciprocal of 0 is ∞ . For all other values the new exponent e_2 is computed with

$$e_2 = -(e_1 - \text{bias}) + \text{bias} = 2 \times \text{bias} - e_1.$$

Finally, we build the reciprocal floating-point number by the concatenation of the sign, exponent, and magnitude.

Figure 2.33 shows the reciprocal in the (1,6,5) floating-point format of the following values:

- 1) $-1/2 = -0.5_{10} = -1.0_2 \times 2^{30-\text{bias}}$
- 2) $1/1.25_{10} = 0.8_{10} \approx (32+19)/64 = 1.10011_2 \times 2^{30-\text{bias}}$
- 3) $1/1.031 = 0.9697_{10} \approx (32+30)/64 = 1.11110_2 \times 2^{30-\text{bias}}$
- 4) $1.0/0 = \infty$
- 5) $1/\infty = 0.0$

Rows one to four show the input floating-point number **a** and the three parts: sign, exponent, and mantissa. Rows five to eight show the reciprocal **r** and the decomposition in the three parts. Note that for the division by zero the transcript window reports: RECIPROCAL: Floating Point divide by zero

2.7.7 Floating-Point Operation Synthesis

Implementing floating-point operations in HDL can become a labor intensive task if we try to build our own complete library including all necessary operations and conversion functions. Luckily, at least for VHDL we have seen the introduction of a very sophisticated library for operations and functions that can be used. This is part of the VHDL-2008 standard and a library with over 7K line of code that is compatible with VHDL-1993 has been provided by David Bishop and can be downloaded from www.eda.org/fphdl. Since most vendors support only a subset of standard VHDL language, small modified versions are available on that webpage that have been tested for Altera, Xilinx, Synopsys, Cadence, and MentorGraphics (ModelSim) tools. The new floating-point standard is documented in Appendix G (pp. 537–549) of the VHDL-2008 LRM and several textbook now cover this new floating-point data types and the operations too [63–65]. To use the library at a minimum in VHDL-1993 we would write

```
LIBRARY ieee_proposed;
USE ieee_proposed.fixed_float_types.ALL;
USE ieee_proposed.float_pkg.ALL;
```

The library allows us to use standard operators like we use for INTEGER and STD_LOGIC_VECTOR data types:

Arithmetic	$+, -, *, /, \text{ABS}, \text{REM}, \text{MOD}$
Logical	NOT, AND, NAND, OR, NOR, XOR, XNOR
Comparison	$=, /=, >, <, \geq, \leq$
Conversion	TO_SLV, TO_SFIXED, TO_FLOAT
Others	RESIZE, SCALB, LOGB, MAXIMUM, MINIMUM

There are also a few predefined constant values. The six values are: zero = zerofp, NaN = nanfp, quite NaN = qnanfp, $\infty = \text{pos_inffp}$, $-\infty = \text{neg_inffp}$, $-0 = \text{neg_zerofp}$. Predefined types of length 32, 64, and 128 as in the IEEE standards 854 and 754 are called FLOAT32, FLOAT64, and FLOAT128, respectively.

Assuming now we like to implement a floating-point number with one sign, six exponent and five fractional bits as in Example 2.15 (p. 75) we would define

```
SIGNAL a, b : FLOAT(6 DOWNTO -5);
SIGNAL s, p : FLOAT(6 DOWNTO -5);
```

and operations can be specified simply as

```
s <= a + b;
p <= a * b;
```

The code is short since left and right sides use the same data type. No scaling or resizing is required. However, the underlying arithmetic uses the default configuration setting that can be seen from the `fixed_float_types` library file. The rounding style is `round_nearest`, `denormalize` and `error_check` are set to true, and three guard bits are used. The minimum HW effort on the other end will happen if we set rounding to `round_zero` (i.e., truncation), `denormalize` and `error_check` false, and guard bits to 0, so basically the opposite to the default setting. Most operations in VHDL-2008 float type are also available in a function form, e.g., for arithmetic function we can use ADD, SUBTRACT, MULTIPLY, DIVIDE, REMAINDER, MODULO, RECIPROCAL, MAC, and SQRT. Then it is much easier to modify the rounding style and guard bits:

```
r <= ADD(l=>a, r=> b, -- Should be the "cheapest" design
          round_style => round_zero,
          guard => 0,
          check_error => false,
          denormalize => false);
```

The left and right operands are specified first, followed by the four synthesis parameters that should give the smallest number of LEs. Note that the IEEE VHDL-2008-1076 (p. 540) says “guard_bits” not “guard” as has been used in the library written by David Bishop.

Comparison can also be used as a function call (similar names as in FORTRAN) via EQ, NE, GT, LT, GE, and LE. For scaling the function SCALB(y,n) implements the operation $y * 2^n$ with a reduced HW effort compared to normal multiplication or divide. MAXIMUM, MINIMUM, square root SQRT, and multiply-and-add MAC are additional functions that can be useful in DSP.

Now let us look at how these functions work in a small example. Since most simulators so far not fully support the new data types such as a negative index in arrays, it seems to be a good approach that we stay with the standard STD_LOGIC_VECTOR as I/O type. The library provides so-called bit-preserving conversion functions that just redefine the meaning of the bits in the STD_LOGIC vector to an sfixed or float type of the same length. Such a conversion is done by the VHDL preprocessor and should *not* consume any hardware resources. On the other hand we also need a *value preserving* operation if we do a conversion between the sfixed and float types. This conversion will preserve the value of our data, but that will indeed require substantiable hardware resources. Now let us have a look at the 32-bit floating-point unit (FPU) that performs some basic operations as well as data conversion.

Example 2.22: A 32-bit Floating-Point Arithmetic Unit

```

Generate Test Data <c> 2013 Uwe Meyer-Baese
For f1=n/d enter n: 1
For f1=n/d enter d: 3
For f2=n/d enter n: 2
For f2=n/d enter d: 3
***** C Float Type Test Data:
* F[1] = 3e aa aa ab
* F[2] = 3f 2a aa ab
0.333333 + 0.666667 = 1
* F[3] = 3f 80 00 00
0.333333 - 0.666667 = -0.333333
* F[3] = be aa aa ab
0.333333 * 0.666667 = 0.222222
* F[3] = 3e 63 8e 3a
0.333333 / 0.666667 = 0.5
* F[3] = 3f 00 00 00
1 / 0.333333 = 3
* F[3] = 40 40 00 00

```

Fig. 2.34. Test data computation using the fp_ops.exe program

The VHDL design file `fpu.vhd`⁸ is shown below:

```

-- Title: Floating-Point Unit
-- Description: This is an arithmetic unit to
-- implement basic 32-bit FP operations.
-- It uses VHDL2008 operations that are also
-- available for 1076-1993 as library function from
-- www.eda.org/fphdl
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
PACKAGE n_bit_int IS                         -- User defined types
    SUBTYPE SLV4 IS STD_LOGIC_VECTOR(3 DOWNTO 0);
    SUBTYPE SLV32 IS STD_LOGIC_VECTOR(31 DOWNTO 0);
END n_bit_int;
LIBRARY work; USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL; USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

LIBRARY ieee_proposed;
USE ieee_proposed.fixed_float_types.ALL;
USE ieee_proposed.fixed_pkg.ALL;
USE ieee_proposed.float_pkg.ALL;
-----
ENTITY fpu IS
    PORT(sel      : IN  SLV4;      -- FP operation number
         dataa   : IN  SLV32;     -- First input
         datab   : IN  SLV32;     -- Second input
         n       : IN  INTEGER;   -- Scale factor 2**n
         result  : OUT SLV32);   -- System output
END;
-----
ARCHITECTURE fpga OF fpu IS

```

⁸ The equivalent Verilog code for this example cannot be composed since the Verilog language currently does not have floating-point library support.

```

-- OP Code of instructions:
-- CONSTANT fix2fp : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"0";
-- CONSTANT fp2fix : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"1";
-- CONSTANT add    : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"2";
-- CONSTANT sub    : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"3";
-- CONSTANT mul    : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"4";
-- CONSTANT div    : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"5";
-- CONSTANT rec    : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"6";
-- CONSTANT scale  : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"7";

TYPE OP_TYPE IS (fix2fp, fp2fix, add, sub, mul, div, rec,
                  scale);

SIGNAL op : OP_TYPE;
SIGNAL a, b, r : FLOAT32;
SIGNAL sfixeda, sfixedr : SFIXED(15 DOWNTO -16);

BEGIN

-- Redefine SLV bit as FP number
  a <= TO_FLOAT(dataa, a);
  b <= TO_FLOAT(datab, b);
-- Redefine SLV bit as 16.16 sfixed number
  sfixeda <= TO_SFIXED(dataa, sfixeda);

P1: PROCESS (a, b, sfixedr, sfixeda, sel, r, n, op)
BEGIN
  r <= (OTHERS => '0'); sfixedr <= (OTHERS => '0');
  CASE CONV_INTEGER(sel) IS
    WHEN 0 =>   r <= TO_FLOAT(sfixeda, r); op <= fix2fp;
    WHEN 1 =>   sfixedr <= TO_SFIXED(a, sfixedr);
                  op <= fp2fix;
    WHEN 2 =>   r <= a + b; op <= add;
    WHEN 3 =>   r <= a - b; op <= sub;
    WHEN 4 =>   r <= a * b; op <= mul;
    WHEN 5 =>   r <= a / b; op <= div;
    WHEN 6 =>   r <= reciprocal(arg=> a); op <= rec;
    WHEN 7 =>   r <= scalb(y=>a, n=>n); op <= scale;
    WHEN OTHERS => op <= scale;
  END CASE;
  -- Interpret FP or 16.16 sfixed bits as SLV bit vector
  IF op = fp2fix THEN
    result <= TO_SLV(sfixedr);
  ELSE
    result <= TO_SLV(r);
  END IF;
END PROCESS P1;

END fpga;

```

First the necessary libraries are called up. Note that the `float` and `fixed` packages need to be downloaded first from www.eda.org/fphdl if your tool does not support the VHDL-2008 types. The ENTITY then includes the selection of our operation, the two input vectors, the scale factor `n`, and the result. Data in 32-bit `float` and 16.16 `sfixed` are then defined. The signal

op is used to display the current operation in plain text in the simulation. The actual floating-point arithmetic unit is placed in the **PROCESS** environment **P1**. The first operation implemented is an **sfixed** to **FLOAT32** conversion followed by **FLOAT32** to **sfixed**. Then come the basic four arithmetic operations: $+$, $-$, $*$, $/$. The operation six is the reciprocal that has only one input. Number seven is the scale operation that implements power-of-two multiply and divide. The last IF statement is used to make a bit preserving conversion (a.k.a. redefinition of the bits) to the **SLV** type for I/O and display. Only for selection one do we have a **sfixed** type; for all the others the output is of type **FLOAT32**.

The design uses 8112 LEs and seven embedded multipliers. A registered performance cannot be measured since registers are not used.

To simulate we first use a small test program in **C** or **MATLAB** to compute some test bench data. In **MATLAB** we can use the format **%tX** and **%bx** to display 32- and 64-bit floats as hex, respectively. For instance if we set **x=1/3** and then type

```
str=sprintf(' FLOAT32 := X\"%tX\"; -- %f',x,x);disp(str)
in the MATLAB prompt window the tool will produce
```

```
FLOAT32 := X"3EAAAAAAB"; -- 0.333333
```

The book CD includes a small program called **fp_ops.exe** that computes test data for 32 and 64 float basic arithmetic operations. If we enter inputs $1/3$ and $2/3$ as **a** and **b** we will get the listing shown in Fig. 2.34 that provides test data for add, subtract, multiply, divide, and reciprocal. To test the input conversion we use the decimal value one that is 0001.0000 in **sfixed** and 3F800000 in hex code for **FLOAT32**. For the scale operation we use a scale by 2^1 , i.e., $1/3 * 2 = 2/3$. The overall simulation is shown in Fig. 2.35. 2.22

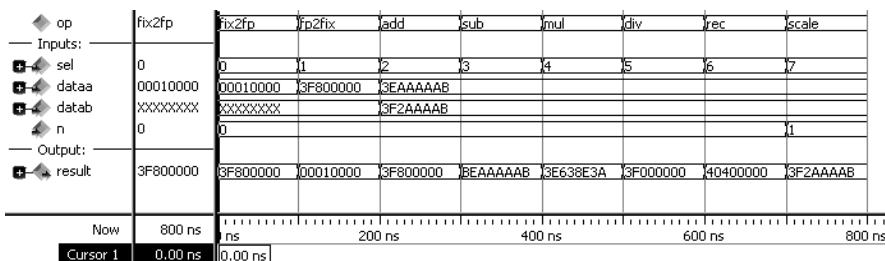


Fig. 2.35. Simulation of eight functions of the floating-point arithmetic unit **fpu**

2.7.8 Floating-Point Synthesis Results

The VHDL-2008 library allows us to write efficient compact code for any float specified. The only disadvantage is that the overall speed of such a design will not be very high since substantial arithmetic is used, but no pipelining is implemented. FPGA vendor usually provide predefined floating-point blocks in 32 and 64 bits that are highly pipelined. Xilinx offers the

Table 2.10. Pipelining in the Altera LPM block library for 32-bit floating-point data

Block	Pipeline range	Default
int/fix to fp	6	6
fp to int/ fix	6	6
add/sub	7 ... 14	11
multiply	5,6,10,11	5
divide	6,14,33	33
inv	20	20

LOGICORE floating-point IP and Altera has a full set of LPM functions. The LPM blocks can be used as graphical block or instantiated from the component library that can be found under `quartus → libraries → vhdl → altera_mf_components.vhd`.

Let us take a brief look at the pipeline requirements to achieve high throughput. Table 2.10 lists the available range and the default setting for Altera's LPM [20]. Figure 2.36 shows a comparison of the throughput that can be achieved when using the VHDL-2008 library and the LPM blocks. The benefit of the pipelining in the LPM blocks is clearly visible. However, keep in mind that many systems we will discuss have feedback, and pipelining is then usually not possible. Then the LPM blocks cannot be used, unless we use an FSM and “wait” until the computation is completed. In order to measure the registered performance **Fmax** with the VHDL-2008 library, registers were added to the input and output ports, but no pipelining inside the block has been used.

Figure 2.37 shows the synthesis results for all eight basic building blocks in 32 and 64 bit width. The solid line shows the required LEs and multipliers with default VHDL-2008 settings, while the dashed lines show the results when synthesis options are set to minimum HW effort, i.e., rounding to `round_zero` (i.e., truncation), `denormalize` and `error_check` false, and guard bits to 0. Conversion between `fixed` and `FLOAT32` requires about 400 LEs. We see that basic math operations `+, -, *` require about 1K LEs with standard setting and twice that for the divide. Note that division and reciprocal in the LPM functions use a substantial number of multipliers and one M9K memory block (not shown in the plot), while the equivalent VHDL-2008 operations do not use any memory or multiplier. The SCALB to multiply or divide by power-of-two is substantially cheaper than the equivalent multiply or divide.

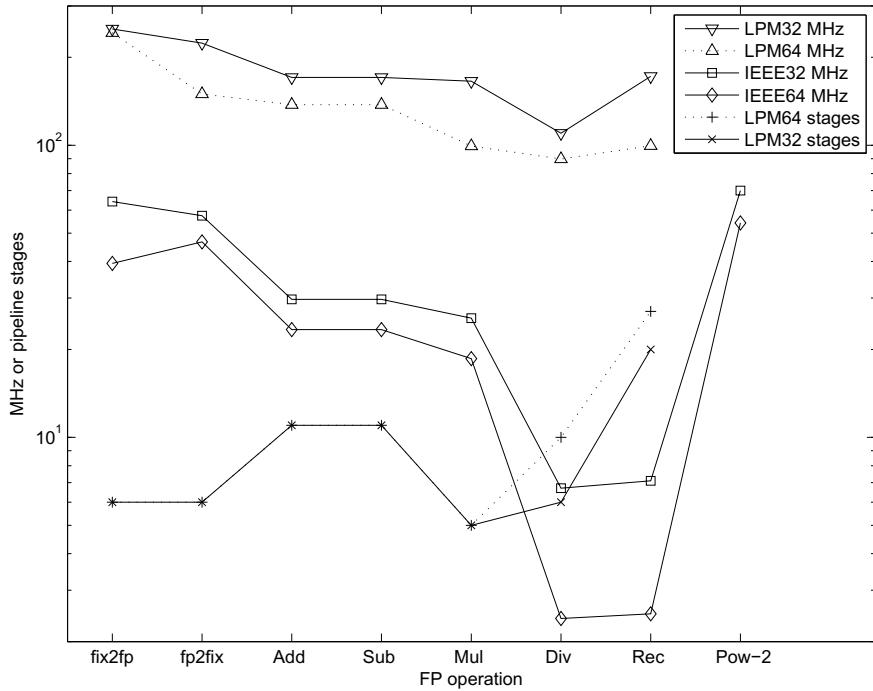


Fig. 2.36. Speed data for VHDL-2008 operations and LPM blocks from Altera

2.8 Multiply-Accumulator (MAC) and Sum of Product (SOP)

DSP algorithms are known to be multiply-accumulate (MAC) intensive. To illustrate, consider the linear convolution sum given by

$$y[n] = f[n] * x[n] = \sum_{k=0}^{L-1} f[k]x[n-k] \quad (2.45)$$

requiring L consecutive multiplications and $L - 1$ addition operations per sample $y[n]$ to compute the sum of products (SOPs). This suggests that an $N \times N$ -bit multiplier needs to be fused together with an accumulator; see Fig. 2.38a. A full-precision $N \times N$ -bit product is $2N$ bits wide. If both operands are (symmetric) signed numbers, the product will only have $2N - 1$ significant bits, i.e., two sign bits. The accumulator, in order to maintain sufficient dynamic range, is often designed to be an extra K bits in width, as demonstrated in the following example.

Example 2.23: The Analog Devices PDSP family ADSP21xx contains a 16×16 array multiplier and an accumulator with an extra 8 bits (for a total accumulator width of $32 + 8 = 40$ bits). With this eight extra bits, at least 2^8 accumulations are

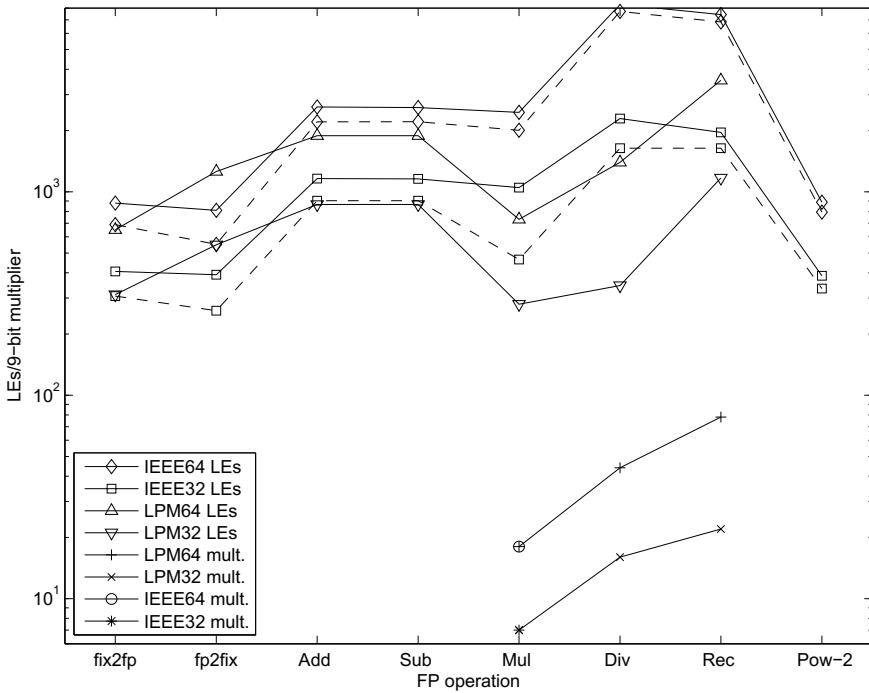


Fig. 2.37. Size data for VHDL-2008 operations and LPM blocks from Altera for default setting (solid line) and minimum hardware effort (dashed line)

possible without sacrificing the output. If both operands are symmetric signed, 2^9 accumulation can be performed. In order to produce the desired output format, such modern PDSPs also include a barrelshifter, which allows the desired adjustment within one clock cycle.

2.23

This overflow consideration in fixed-point PDSP is important to mainstream digital signal processing, which requires that DSP objects be computed in real time without unexpected interruptions. Recall that checking and servicing accumulator overflow interrupts the data flow and carries a significant temporal liability. By choosing the number of guard bits correctly, the liability can be eliminated.

An alternative approach to the MAC of a conventional PDSP for computing an SOP will be discussed in the next section.

2.8.1 Distributed Arithmetic Fundamentals

Distributed arithmetic (DA) is an important FPGA technology. It is extensively used in computing the sum of products:

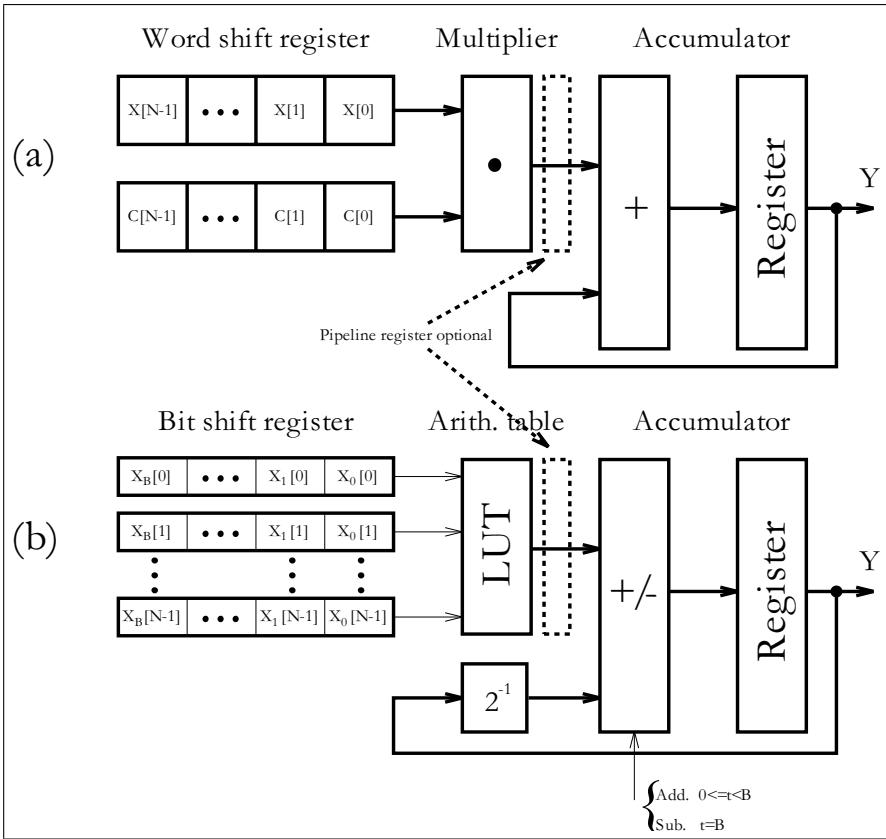


Fig. 2.38. (a) Conventional PDSP and (b) Shift-Adder DA Architecture

$$y = \langle c, x \rangle = \sum_{n=0}^{N-1} c[n] \times x[n]. \quad (2.46)$$

Besides convolution, correlation, DFT computation, and the RNS inverse mapping discussed earlier can also be formulated as such SOPs. Completing a filter cycle, when using a conventional arithmetic unit, would take approximately N MAC cycles. This can be shortened with pipelining but can, nevertheless, be prohibitively long. This is a fundamental problem when general-purpose multipliers are used.

In many DSP applications, a general-purpose multiplication is technically not required. If the filter coefficients $c[n]$ are known a priori, then technically the partial product term $c[n]x[n]$ becomes a multiplication with a constant (i.e., scaling). This is an important difference and is a prerequisite for a DA design.

The first discussion of DA can be traced to a 1973 paper by Croisier [66] and DA was popularized by Peled and Liu [67]. Yiu [68] extended DA to signed numbers, and Kammeyer [69] and Taylor [70] studied quantization effects in DA systems. DA tutorials are available from White [71] and Kammeyer [72]. DA also is addressed in textbooks [73, 74]. To understand the DA design paradigm, consider the “sum of products” inner product shown below:

$$\begin{aligned} y &= \langle \mathbf{c}, \mathbf{x} \rangle = \sum_{n=0}^{N-1} c[n] \times x[n] \\ &= c[0]x[0] + c[1]x[1] + \dots + c[N-1]x[N-1]. \end{aligned} \quad (2.47)$$

Assume further that the coefficients $c[n]$ are known constants and $x[n]$ is a variable. An unsigned DA system assumes that the variable $x[n]$ is represented by:

$$x[n] = \sum_{b=0}^{B-1} x_b[n] \times 2^b \quad \text{with } x_b[n] \in [0, 1], \quad (2.48)$$

where $x_b[n]$ denotes the b^{th} bit of $x[n]$, i.e., the n^{th} sample of \mathbf{x} . The inner product y can, therefore, be represented as:

$$y = \sum_{n=0}^{N-1} c[n] \times \sum_{b=0}^{B-1} x_b[n] \times 2^b. \quad (2.49)$$

Redistributing the order of summation (thus the name “distributed arithmetic”) results in:

$$\begin{aligned} y &= c[0] (x_{B-1}[0]2^{B-1} + x_{B-2}[0]2^{B-2} + \dots + x_0[0]2^0) \\ &\quad + c[1] (x_{B-1}[1]2^{B-1} + x_{B-2}[1]2^{B-2} + \dots + x_0[1]2^0) \\ &\quad \vdots \\ &\quad + c[N-1] (x_{B-1}[N-1]2^{B-1} + \dots + x_0[N-1]2^0) \\ &= (c[0]x_{B-1}[0] + c[1]x_{B-1}[1] + \dots + c[N-1]x_{B-1}[N-1]) 2^{B-1} \\ &\quad + (c[0]x_{B-2}[0] + c[1]x_{B-2}[1] + \dots + c[N-1]x_{B-2}[N-1]) 2^{B-2} \\ &\quad \vdots \\ &\quad + (c[0]x_0[0] + c[1]x_0[1] + \dots + c[N-1]x_0[N-1]) 2^0, \end{aligned}$$

or in more compact form

$$y = \sum_{b=0}^{B-1} 2^b \times \sum_{n=0}^{N-1} \underbrace{c[n] \times x_b[n]}_{f(c[n], x_b[n])} = \sum_{b=0}^{B-1} 2^b \times \sum_{n=0}^{N-1} f(c[n], x_b[n]). \quad (2.50)$$

Implementation of the function $f(c[n], x_b[n])$ requires special attention. The preferred implementation method is to realize the mapping $f(c[n], x_b[n])$ using one LUT. That is, a 2^N -word LUT is preprogrammed to accept an N -bit input vector $\mathbf{x}_b = [x_b[0], x_b[1], \dots, x_b[N-1]]$, and output $f(c[n], x_b[n])$. The individual mappings $f(c[n], x_b[n])$ are weighted by the appropriate power-of-two factor and accumulated. The accumulation can be efficiently implemented using a shift-adder as shown in Fig. 2.38b. After N look-up cycles, the inner product y is computed.

Example 2.24: Unsigned DA Convolution

A third order inner product is defined by the inner product equation $y = \langle \mathbf{c}, \mathbf{x} \rangle = \sum_{n=0}^2 c[n]x[n]$. Assume that the 3-bit coefficients have the values $c[0] = 2$, $c[1] = 3$, and $c[2] = 1$. The resulting LUT, which implements $f(c[n], x_b[n])$, is defined below:

$x_b[2]$	$x_b[1]$	$x_b[0]$	$f(c[n], x_b[n])$
0	0	0	$1 \times 0 + 3 \times 0 + 2 \times 0 = 0_{10} = 000_2$
0	0	1	$1 \times 0 + 3 \times 0 + 2 \times 1 = 2_{10} = 010_2$
0	1	0	$1 \times 0 + 3 \times 1 + 2 \times 0 = 3_{10} = 011_2$
0	1	1	$1 \times 0 + 3 \times 1 + 2 \times 1 = 5_{10} = 101_2$
1	0	0	$1 \times 1 + 3 \times 0 + 2 \times 0 = 1_{10} = 001_2$
1	0	1	$1 \times 1 + 3 \times 0 + 2 \times 1 = 3_{10} = 011_2$
1	1	0	$1 \times 1 + 3 \times 1 + 2 \times 0 = 4_{10} = 100_2$
1	1	1	$1 \times 1 + 3 \times 1 + 2 \times 1 = 6_{10} = 110_2$

The inner product, with respect to $x[n] = \{x[0] = 1_{10} = 001_2, x[1] = 3_{10} = 011_2, x[2] = 7_{10} = 111_2\}$, is obtained as follows:

Step t	$x_t[2]$	$x_t[1]$	$x_t[0]$	$f[t]$	$+ACC[t-1]$	$=ACC[t]$
0	1	1	1	$6 \times 2^0 +$	0	= 6
1	1	1	0	$4 \times 2^1 +$	6	= 14
2	1	0	0	$1 \times 2^2 +$	14	= 18

As a numerical check, note that

$$\begin{aligned} y &= \langle \mathbf{c}, \mathbf{x} \rangle = c[0]x[0] + c[1]x[1] + c[2]x[2] \\ &= 2 \times 1 + 3 \times 3 + 1 \times 7 = 18. \checkmark \end{aligned}$$

2.24

For a hardware implementation, instead of shifting each intermediate value by b (which will demand an expensive barrelshifter) it is more appropriate to shift the accumulator content itself in each iteration one bit to the right. It is easy to verify that this will give the same results.

The bandwidth of an N^{th} order B -bit linear convolution, using general-purpose MACs and DA hardware, can be compared. Figure 2.38 shows the architectures of a conventional PDSP and the same realization using distributed arithmetic.

Assume that a LUT and a general-purpose multiplier have the same delay $\tau = \tau(\text{LUT}) = \tau(\text{MUL})$. The computational latencies are then $B\tau(\text{LUT})$ for DA and $N\tau(\text{MUL})$ for the PDSP. In the case of small bit width B , the speed of the DA design can therefore be significantly faster than a MAC-based design. In Chap. 3, comparisons will be made for specific filter design examples.

2.8.2 Signed DA Systems

In the following, we wish to discuss how (2.47) should be modified, in order to process a signed two's complement number. In two's complement, the MSB is used to distinguish between positive and negative numbers. For instance, from Table 2.1 (p. 59) we see that decimal -3 is coded as $101_2 = -4 + 0 + 1 = -3_{10}$. We use, therefore, the following $(B + 1)$ -bit representation

$$x[n] = -2^B \times x_B[n] + \sum_{b=0}^{B-1} x_b[n] \times 2^b. \quad (2.51)$$

Combining this with (2.49), the outcome y is defined by:

$$y = -2^B \times f(c[n], x_B[n]) + \sum_{b=0}^{B-1} 2^b \times \sum_{n=0}^{N-1} f(c[n], x_b[n]). \quad (2.52)$$

To achieve the signed DA system we therefore have two choices to modify the unsigned DA system. They are

- An accumulator with add/subtract control
- Using a ROM with one additional input

Most often the switchable accumulator is preferred, because the additional input bit in the table requires a table with twice as many words. The following example demonstrates the processing steps for the add/sub switch design.

Example 2.25: Signed DA Inner Product

Consider again a third order inner product defined by the convolution sum $y = \langle \mathbf{c}, \mathbf{x} \rangle = \sum_{n=0}^2 c[n]x[n]$. Assume that the data $x[n]$ is given in 4-bit two's complement encoding and that the coefficients are $c[0] = -2$, $c[1] = 3$, and $c[2] = 1$. The corresponding LUT table is given below:

$x_b[2]$	$x_b[1]$	$x_b[0]$	$f(c[k], x_b[n])$
0	0	0	$1 \times 0 + 3 \times 0 - 2 \times 0 = 0_{10}$
0	0	1	$1 \times 0 + 3 \times 0 - 2 \times 1 = -2_{10}$
0	1	0	$1 \times 0 + 3 \times 1 - 2 \times 0 = 3_{10}$
0	1	1	$1 \times 0 + 3 \times 1 - 2 \times 1 = 1_{10}$
1	0	0	$1 \times 1 + 3 \times 0 - 2 \times 0 = 1_{10}$
1	0	1	$1 \times 1 + 3 \times 0 - 2 \times 1 = -1_{10}$
1	1	0	$1 \times 1 + 3 \times 1 - 2 \times 0 = 4_{10}$
1	1	1	$1 \times 1 + 3 \times 1 - 2 \times 1 = 2_{10}$

The values of $x[k]$ are $x[0] = 1_{10} = 0001_{2C}$, $x[1] = -3_{10} = 1101_{2C}$, and $x[2] = 7_{10} = 0111_{2C}$. The output at sample index k , namely y , is defined as follows:

Step t	$x_t[2]$	$x_t[1]$	$x_t[0]$	$f[t] \times 2^t$	$+Y[t-1]=Y[t]$
0	1	1	1	2×2^0	$+ 0 = 2$
1	1	0	0	1×2^1	$+ 2 = 4$
2	1	1	0	4×2^2	$+ 4 = 20$
$x_t[2]$ $x_t[1]$ $x_t[0]$				$f[t] \times (-2^t) + Y[t-1] = Y[t]$	
3	0	1	0	$3 \times (-2^3)$	$+ 20 = -4$

A numerical check results in $c[0]x[0] + c[1]x[1] + c[2]x[2] = -2 \times 1 + 3 \times (-3) + 1 \times 7 = -4 \checkmark$

2.25

2.8.3 Modified DA Solutions

In the following we wish to discuss two interesting modifications to the basic DA concept, where the first variation reduces the size, and the second increases the speed.

If the number of coefficients N is too large to implement the full word with a single LUT (recall that input LUT bit width = number of coefficients), then we can use partial tables and add the results. If we also add pipeline registers, this modification will not reduce the speed, but can dramatically reduce the size of the design, because the size of a LUT grows exponentially with the address space, i.e., the number of input coefficients N . Suppose the length LN inner product

$$y = \langle \mathbf{c}, \mathbf{x} \rangle = \sum_{n=0}^{LN-1} c[n]x[n] \quad (2.53)$$

is to be implemented using a DA architecture. The sum can be partitioned into L independent N^{th} parallel DA LUTs resulting in

$$y = \langle \mathbf{c}, \mathbf{x} \rangle = \sum_{l=0}^{L-1} \sum_{n=0}^{N-1} c[Nl+n]x[Nl+n]. \quad (2.54)$$

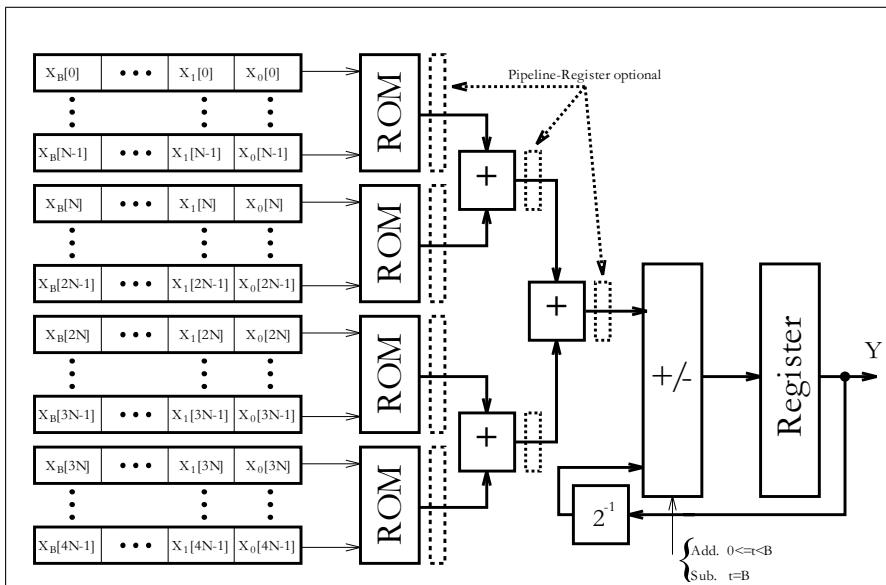


Fig. 2.39. Distributed arithmetic with table partitioning to yield a reduced size

This is shown in Fig. 2.39 for a realization of a $4N$ DA design requiring three postadditional adders. The size of the table is reduced from one $2^{4N} \times B$ LUT to four $2^N \times B$ tables.

Another variation of the DA architecture increases speed at the expense of additional LUTs, registers, and adders. A basic DA architecture, for a length N^{th} sum-of-product computation, accepts one bit from each of N words. If two bits per word are accepted, then the computational speed can be essentially doubled. The maximum speed can be achieved with the fully pipelined word-parallel architecture shown in Fig. 2.40. Here, a new result of a length four sum-of-product is computed for 4-bit signed coefficients at each LUT cycle. For maximum speed, we have to provide a separate ROM (with identical content) for each bit vector $x_b[n]$. But the maximum speed can become expensive: If we double the input bit width, we need twice as many LUTs, adders and registers. If the number of coefficients N is limited to four or eight this modification gives attractive performance, essentially outperforming all commercially available programmable signal processors, as we will see in Chap. 3.

2.9 Computation of Special Functions Using CORDIC

If a digital signal processing algorithm is implemented with FPGAs and the algorithm uses a nontrivial (transcendental) algebraic function, like \sqrt{x} or

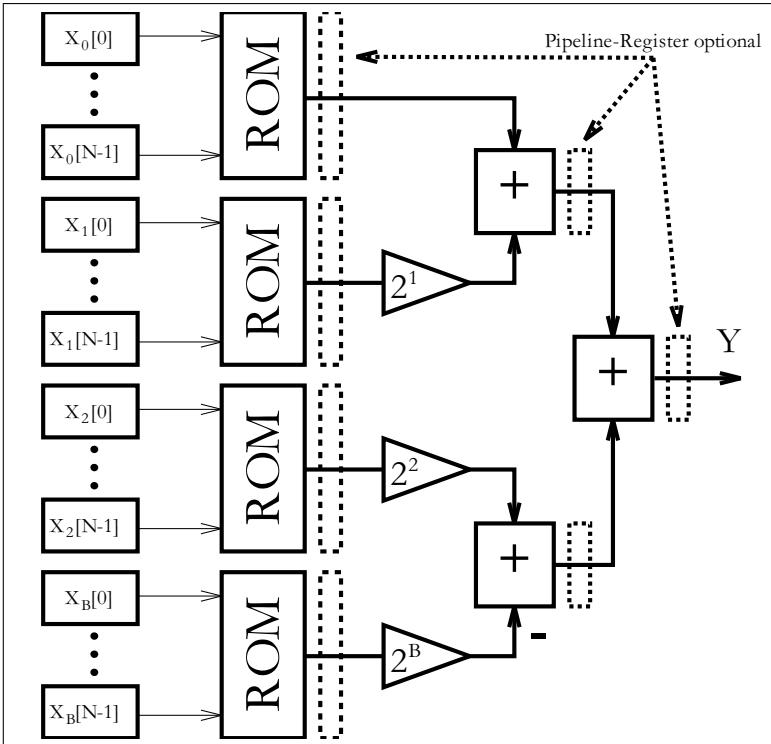


Fig. 2.40. Higher-order distributed arithmetic optimized for speed

arctan y/x , we can always use the Taylor series to approximate this function, i.e.,

$$f(x) = \sum_{k=0}^K \frac{f^k(x_0)}{k!} (x - x_0)^k, \quad (2.55)$$

where $f^k(x)$ is the k^{th} derivative of $f(x)$ and $k! = k \times (k - 1) \dots \times 1$. The problem is then reduced to a sequence of multiply and add operations. A more efficient, alternative approach, based on the *Coordinate Rotation Digital Computer* (CORDIC) algorithm can also be considered. The CORDIC algorithm is found in numerous applications, such as pocket calculators [75], and in mainstream DSP objects, such as adaptive filters, FFTs, DCTs [76], demodulators [77], and neural networks [45]. The basic CORDIC algorithm can be found in two classic papers by Volder [78] and Walther [79]. Some theoretical extensions have been made, such as the extension of range in the hyperbolic mode, or the quantization error analysis by Hu et al. [80], and Meyer-Bäse et al. [77]. VLSI implementations have been discussed in Ph.D. theses, such as those by Timmermann [81] and Hahn [82]. The first FPGA

Table 2.11. CORDIC algorithm modes

Mode	Angle θ_k	Shift sequence	Radius factor
circular $m = 1$	$\tan^{-1}(2^{-k})$	$0, 1, 2, \dots$	$K_1 = 1.65$
linear $m = 0$	2^{-k}	$1, 2, \dots$	$K_0 = 1.0$
hyperbolic $m = -1$	$\tanh^{-1}(2^{-k})$	$1, 2, 3, 4, 4, \dots$	$K_{-1} = 0.80$

implementations were investigated by Meyer-Bäse et al. [4, 77]. The realization of the CORDIC algorithm in distributed arithmetic was investigated by Ma [83]. A very detailed overview including details of several applications, was provided by Hu [76] in a 1992 IEEE Signal Processing Magazine review paper.

The original CORDIC algorithm by Volder [78] computes a multiplier-free coordinate conversion between rectangular (x, y) and polar (R, θ) coordinates. Walther [79] generalized the CORDIC algorithm to include circular ($m = 1$), linear ($m = 0$), and hyperbolic ($m = -1$) transforms. For each mode, two rotation directions are identified. For *vectoring*, a vector with starting coordinates (X_0, Y_0) is rotated in such a way that the vector finally lies on the abscissa (i.e., x axis) by iteratively converging Y_K to zero. For *rotation*, a vector with a starting coordinate (X_0, Y_0) is rotated by an angle θ_0 in such a way that the final value of the angle register, denoted Z , converges to zero. The angle θ_k is chosen so that each iteration can be performed with an addition and a binary shift. Table 2.11 shows, in the second column, the choice for the rotation angle for the three modes $m = 1, 0$, and -1 .

Now we can formally define the CORDIC algorithm as follows:

Algorithm 2.26: CORDIC Algorithm

At each iteration, the CORDIC algorithm implements the mapping:

$$\begin{bmatrix} X_{k+1} \\ Y_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & m\delta_k 2^{-k} \\ \delta_k 2^{-k} & 1 \end{bmatrix} \begin{bmatrix} X_k \\ Y_k \end{bmatrix} \quad (2.56)$$

$$Z_{k+1} = Z_k + \delta_k \theta_k,$$

where the angle θ_k is given in Table 2.11, $\delta_k = \pm 1$, and the two rotation directions are $Z_K \rightarrow 0$ and $Y_K \rightarrow 0$.

This means that six operational modes exist, and they are summarized in Table 2.12. A consequence is that nearly all transcendental functions can be computed with the CORDIC algorithm. With a proper choice of the initial values, the function $X \times Y, Y/X, \sin(Z), \cos(Z), \tan^{-1}(Y), \sinh(Z), \cosh(Z)$, and $\tanh(Z)$ can directly be computed. Additional functions may be generated by choosing appropriate initialization, sometimes combined with multiple modes of operation, as shown in the following listing:

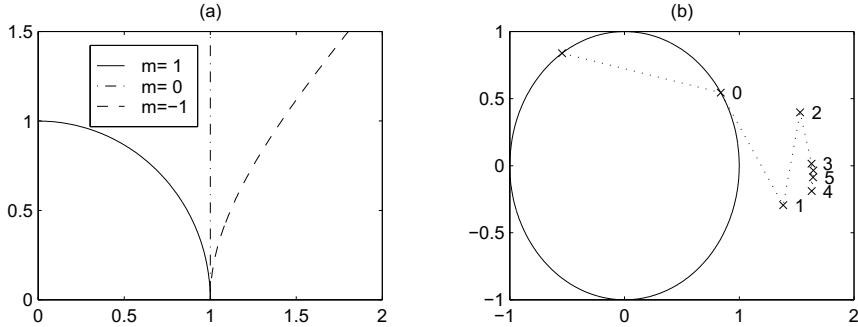


Fig. 2.41. CORDIC. (a) Modes. (b) Example of circular vectoring

$$\begin{aligned}\tan(Z) &= \sin(Z)/\cos(Z) \\ \tanh(Z) &= \sinh(Z)/\cosh(Z) \\ \exp(Z) &= \sinh(Z) + \cosh(Z) \\ \log_e(W) &= 2 \tanh^{-1}(Y/X)\end{aligned}$$

$$\begin{aligned}&\text{Modes: } m=1, 0 \\ &\text{Modes: } m=-1, 0 \\ &\text{Modes: } m=-1; \quad x = y = 1 \\ &\text{Modes: } m=-1 \\ &\quad \text{with } X = W + 1, Y = W - 1 \\ &\text{Modes: } m=-1 \\ &\quad \text{with } X = W + \frac{1}{4}, Y = W - \frac{1}{4}.\end{aligned}$$

A careful analysis of (2.56) reveals that the iteration vectors only approach the curves shown in Fig. 2.41a. The length of the vectors changes with each iteration, as shown in Fig. 2.41b. This change in length does *not* depend on the starting angle and after K iterations the same change (called radius factor) always occurs. In the last column of Table 2.11 these radius factors are shown. To ensure that the CORDIC algorithm converges, the sum of all remaining rotation angles must be larger than the actual rotation angle. This is the case for linear and circular transforms. For the hyperbolic mode, all iterations of the form $n_{k+1} = 3n_k + 1$ have to be repeated. These are the iterations 4, 13, 40, 121

Table 2.12. Modes m of operation for the CORDIC algorithm

m	$Z_K \rightarrow 0$	$Y_K \rightarrow 0$
1	$X_K = K_1(X_0 \cos(Z_0) - Y_0 \sin(Z_0))$ $Y_K = K_1(Y_0 \cos(Z_0) + X_0 \sin(Z_0))$	$X_K = K_1 \sqrt{X_0^2 + Y_0^2}$ $Z_K = Z_0 + \arctan(Y_0/X_0)$
0	$X_K = X_0$ $Y_K = Y_0 + X_0 \times Z_0$	$X_K = X_0$ $Z_K = Z_0 + Y_0/X_0$
-1	$X_K = K_{-1}(X_0 \cosh(Z_0) - Y_0 \sinh(Z_0))$ $Y_K = K_{-1}(Y_0 \cosh(Z_0) + X_0 \sinh(Z_0))$	$X_K = K_{-1} \sqrt{X_0^2 - Y_0^2}$ $Z_K = Z_0 + \tanh^{-1}(Y_0/X_0)$

Output precision can be estimated using a procedure developed by Hu [84] and illustrated in Fig. 2.42. The graph shows the effective bit precision for the circular mode, depending on the X, Y path width, and the number of iterations. If b bits is the desired output precision, the “rule of thumb” suggests that the X, Y path should have $\log_2(b)$ additional guard bits. From Fig. 2.43, it can also be seen that the bit width of the Z path should have the same precision as that for X and Y .

In contrast to the circular CORDIC algorithm, the effective resolution of a hyperbolic CORDIC cannot be computed analytically because the precision depends on the angular values of $z(k)$ at iteration k . Hyperbolic precision can, however, be estimated using simulation. Figure 2.44 shows the minimum accuracy estimate computed over 1000 test values for each bit-width/number combination of the possible iterations. The 3D representation shows the number of iterations, the bit width of the X/Y path, and the resulting minimum precision of the result in terms of effective bits. The contour lines allow an exchange between the number of iterations and the bit width. For example, to achieve 10-bit precision, one can use a 21-bit X/Y path and 18 iterations, or 14 iterations at 24 bits.

2.9.1 CORDIC Architectures

Two basic structures are used to implement a CORDIC architecture: the more compact state machine or the high-speed, fully pipelined processor.

If computation time is not critical, then a state machine as shown in Fig. 2.45 is applicable. In each cycle, exactly one iteration of (2.56) will be computed. The most complex part of this design is the two barrelshifters. The

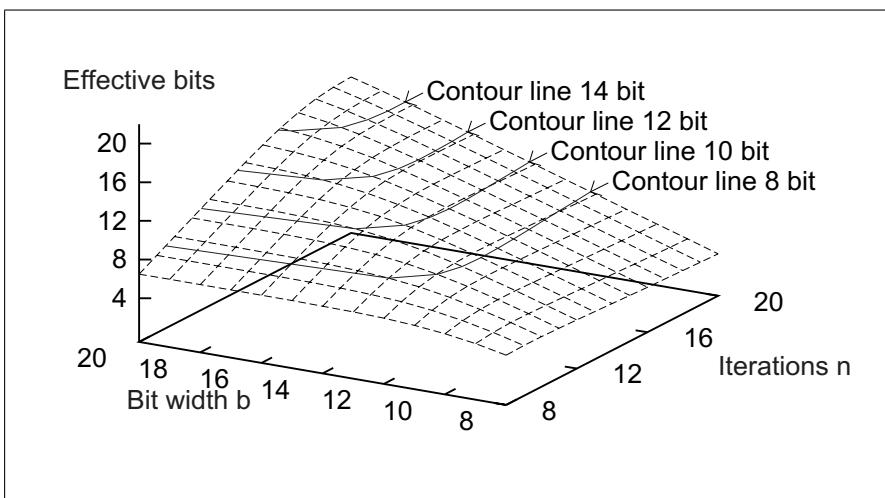
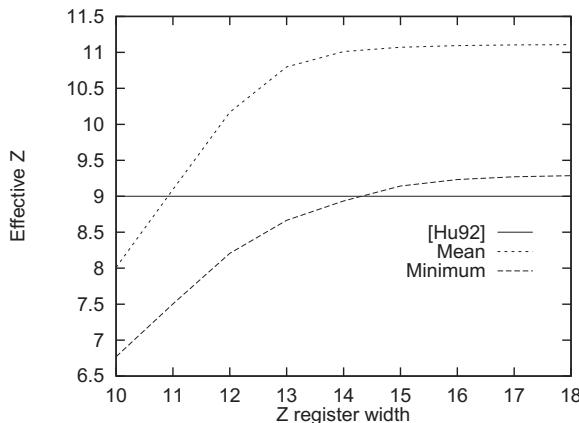
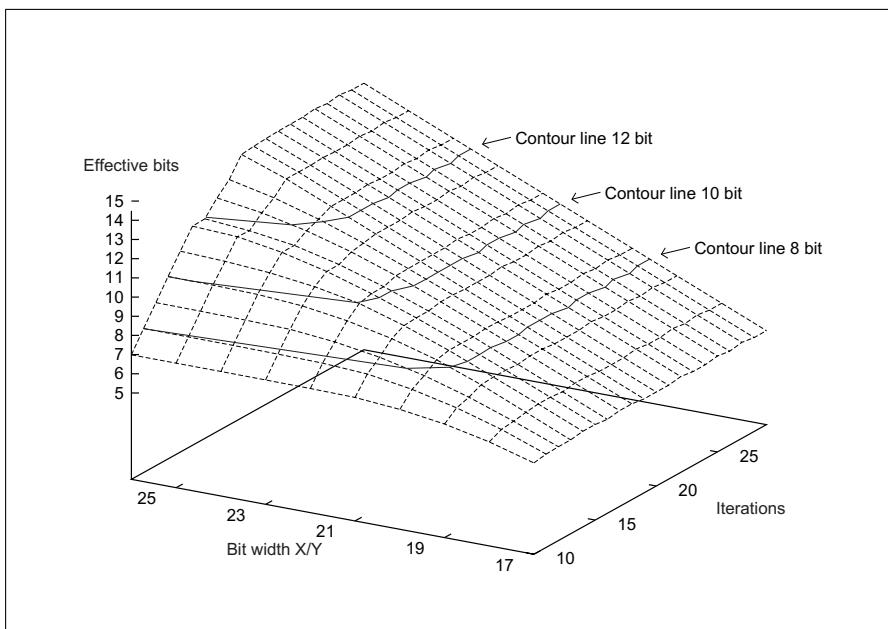


Fig. 2.42. Effective bits in circular mode

**Fig. 2.43.** Resolution of phase for circular mode**Fig. 2.44.** Effective bits in hyperbolic mode

two barrelshifters can be replaced by a single barrelshifter, using a multiplexer as shown in Fig. 2.46, or a serial (right, or right/left) shifter. Table 2.13 compares different design options for a 13-bit implementation using Xilinx XC3K FPGAs.

If high speed is needed, a fully pipelined version of the design shown in Fig. 2.47 can be used. Figure 2.47 shows eight iterations of a circular

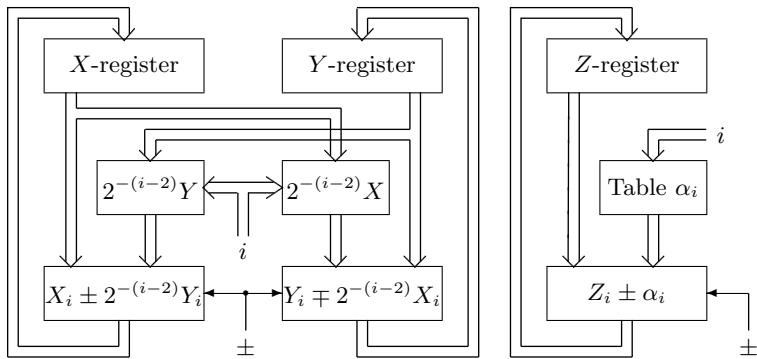


Fig. 2.45. CORDIC state machine

Table 2.13. Effort estimation for a CORDIC machine with 13-bits plus sign for X/Y path. (Abbreviations: Ac=accumulator; BS=barrelshifter; RS=serial right shifter; LRS=serial left/right shifter)

Structure	Registers	Multiplexer	Adder	Shifter	\sum LE	Cycle
2BS+2Ac	2×7	0	2×14	2×19.5	81	12
2RS+2Ac	2×7	0	2×14	2×6.5	55	46
2LRS+2Ac	2×7	0	2×14	2×8	58	39
1BS+2Ac	7	3×7	2×14	19.5	75.5	20
1RS+2Ac	7	3×7	2×14	6.5	62.5	56
1LRS+2Ac	7	3×7	2×14	8	64	74
1BS+1Ac	3×7	2×7	14	19.5	68.5	20
1RS+1Ac	3×7	2×7	14	6.5	55.5	92
1LRS+1Ac	3×7	2×7	14	8	57	74

CORDIC. After an initial delay of K cycles, a new output value becomes available after each cycle. As with array multipliers, CORDIC implementations have a quadratic growth in LE complexity as the bit width increases (see Fig. 2.47).

The following example shows the first four steps of a circular-vectorizing fully pipelined design.

Example 2.27: Circular CORDIC in Vectoring Mode

The first iteration rotates the vectors from the second or third quadrant to the first or fourth, respectively. The shift sequence is 0,0,1, and 2. The rotation angle of the first four steps becomes: $\arctan(\infty) = 90^\circ$, $\arctan(2^0) =$

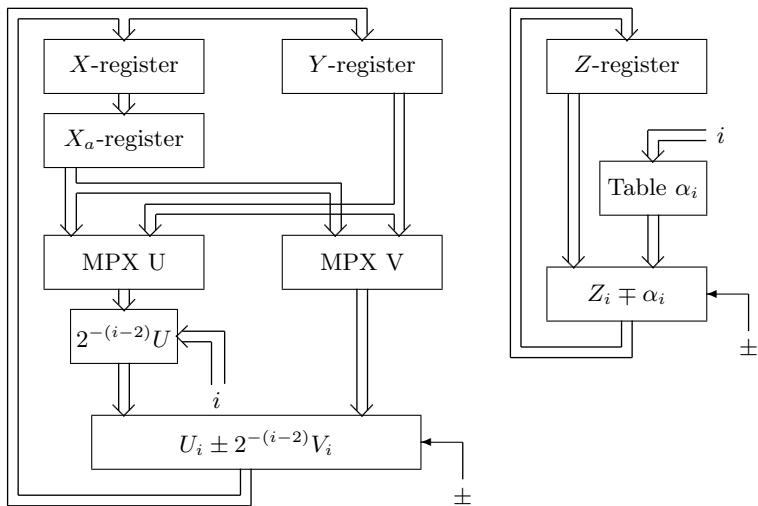


Fig. 2.46. CORDIC machine with reduced complexity

45° , $\arctan(2^{-1}) = 26.5^\circ$, and $\arctan(2^{-2}) = 14^\circ$. The VHDL code⁹ for 8-bit data can be implemented as follows:

```

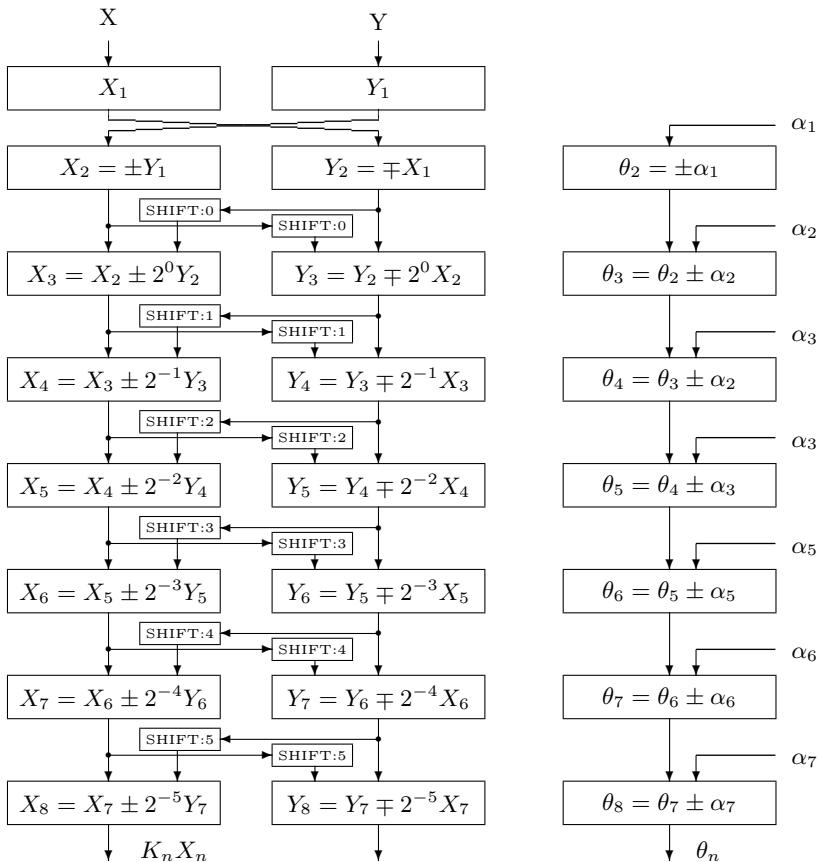
PACKAGE n_bit_int IS      -- User defined types
    SUBTYPE S8 IS INTEGER RANGE -128 TO 127;
    SUBTYPE S9 IS INTEGER RANGE -256 TO 256;
    TYPE AO_3S9 IS ARRAY (0 TO 3) OF S9;
END n_bit_int;

LIBRARY work;
USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
-----
ENTITY cordic IS           -----> Interface
    PORT (clk   : IN STD_LOGIC; -- System clock
          reset : IN STD_LOGIC; -- Asynchronous reset
          x_in  : IN S8;    -- System real or x input
          y_in  : IN S8;    -- System imaginary or y input
          r     : OUT S9;   -- Radius result
          phi   : OUT S9;   -- Phase result
          eps   : OUT S9); -- Error of results
END cordic;
-----
ARCHITECTURE fpga OF cordic IS

```

⁹ The equivalent Verilog code `cordic.v` for this example can be found in Appendix A on page 804. Synthesis results are shown in Appendix B on page 881.

**Fig. 2.47.** Fast CORDIC pipeline

```
--SIGNAL x, y, z : A0_3S9; -- Array of Bytes
BEGIN

P1: PROCESS(x_in, y_in, reset, clk) --> Behavioral Style
    VARIABLE x, y, z : A0_3S9; -- Array of Bytes
BEGIN
    IF reset = '1' THEN -- Asynchronous clear
        FOR K IN 0 TO 3 LOOP
            x(k) := 0; y(k) := 0; z(k) := 0;
        END LOOP;
        r <= 0; eps <= 0; phi <= 0;
    ELSIF rising_edge(clk) THEN
        r <= x(3);                      -- Compute last value first in
        phi <= z(3);                    -- sequential VHDL statements !!
        eps <= y(3);

        IF y(2) >= 0 THEN             -- Rotate 14 degrees
            ...
        ELSE
            ...
        END IF;
    END IF;
END PROCESS P1;
```

```

x(3) := x(2) + y(2) /4;
y(3) := y(2) - x(2) /4;
z(3) := z(2) + 14;
ELSE
  x(3) := x(2) - y(2) /4;
  y(3) := y(2) + x(2) /4;
  z(3) := z(2) - 14;
END IF;

IF y(1) >= 0 THEN          -- Rotate 26 degrees
  x(2) := x(1) + y(1) /2;
  y(2) := y(1) - x(1) /2;
  z(2) := z(1) + 26;
ELSE
  x(2) := x(1) - y(1) /2;
  y(2) := y(1) + x(1) /2;
  z(2) := z(1) - 26;
END IF;

IF y(0) >= 0 THEN          -- Rotate 45 degrees
  x(1) := x(0) + y(0);
  y(1) := y(0) - x(0);
  z(1) := z(0) + 45;
ELSE
  x(1) := x(0) - y(0);
  y(1) := y(0) + x(0);
  z(1) := z(0) - 45;
END IF;

-- Test for x_in < 0 rotate 0,+90, or -90 degrees
IF x_in >= 0 THEN
  x(0) := x_in;           -- Input in register 0
  y(0) := y_in;
  z(0) := 0;
ELSIF y_in >= 0 THEN
  x(0) := y_in;
  y(0) := - x_in;
  z(0) := 90;
ELSE
  x(0) := - y_in;
  y(0) := x_in;
  z(0) := -90;
END IF;
END IF;
END PROCESS;

END fpga;

```

Figure 2.48 shows the simulation of the conversion of $X_0 = -41$, and $Y_0 = 55$. Note that the radius is enlarged to $R = X_K = 111 = 1.618\sqrt{X_0^2 + Y_0^2}$ and the accumulated angle in degrees is $\arctan(Y_0/X_0) = 123^\circ$. The design requires 276 LEs and runs with a Speed synthesis optimization at 209.6 MHz using no embedded multiplier.

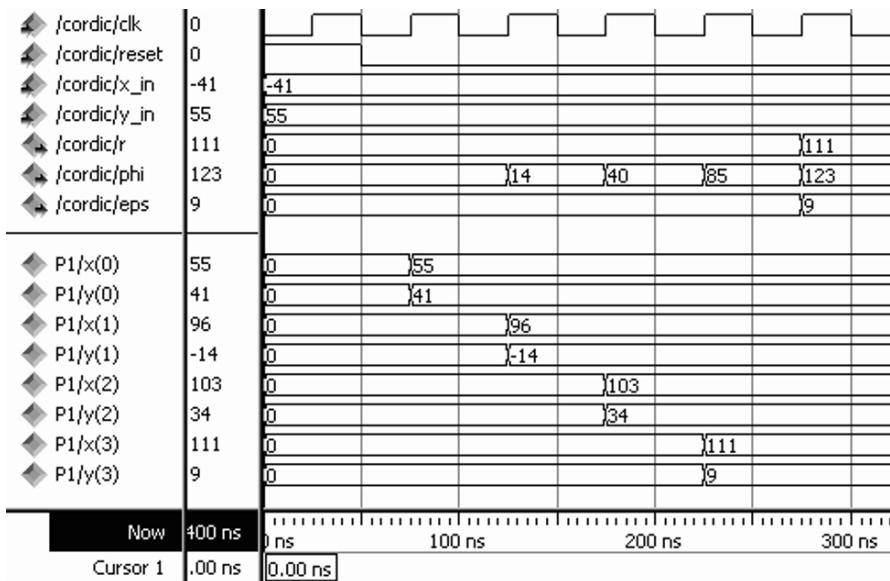


Fig. 2.48. CORDIC simulation results

The actual LE count in the previous example is larger than that expected for a four-stage 8-bit pipeline design that is $5 \times 8 \times 3 = 120$ LEs. The increase by a factor of two comes from the fact that a FPGA uses an N -bit switchable adder/subtractor that needs $2N$ LEs. It needs $2N$ LEs because the LE has only three inputs in the fast arithmetic mode, and the switch mode needs four input LUTs. An ALM type LE, see Fig. 1.7a, p. 12, would be needed, with at least four inputs per LE, to reduce the count by a factor of two.

2.10 Computation of Special Functions using MAC Calls

The CORDIC algorithm introduced in the previous section allows one to implement a wide variety of functions at a moderate implementation cost. The only disadvantage is that some high-precision functions need a large number of iterations, because the number of bits is linearly proportional to the number of iterations. In a pipelined implementation this results in a large latency.

With the advent of fast embedded array multipliers in new FPGA families like Spartan or Cyclone, see Table 1.4 (p. 11), the implementation of special functions via a polynomial approximation has becomes a viable option. We have introduced the Taylor series approximation in (2.55), p. 132. The Taylor series approximation converges fast for some functions, e.g., $\exp(x)$, but

needs many product terms for some other special functions, e.g., $\arctan(x)$, to approximate with sufficient precision. In these cases a Chebyshev approximation can be used to shorten the number of iterations or product terms required.

2.10.1 Chebyshev Approximations

The Chebyshev approximation is based on the Chebyshev polynomial

$$T_k(x) = \cos(k \times \arccos(x)) \quad (2.57)$$

defined for the range $-1 \leq x \leq 1$. The $T_k(x)$ may look like trigonometric functions, but using some algebraic identities and manipulations allow us to write (2.57) as a true polynomial. The first few polynomials look like

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ T_5(x) &= 16x^5 - 20x^3 + 5x \\ T_6(x) &= 32x^6 - 48x^4 + 18x^2 - 1 \\ &\vdots \end{aligned} \quad (2.58)$$

In [85] we find a list of the first 12 polynomials. The first six polynomials are graphical interpreted in Fig. 2.49. In general, Chebyshev polynomials obey the following iterative rule

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x) \quad \forall k \geq 2. \quad (2.59)$$

A function approximation can now be written as

$$f(x) = \sum_{k=0}^{N-1} c(k)T_k(x). \quad (2.60)$$

Because all discrete Chebyshev polynomials are orthogonal to each other it follows that forward and inverse transform are unique, i.e., bijective [86, p. 191]. The question now is why (2.60) is so much better than, for instance, a polynomial using the Taylor approximation (2.55)

$$f(x) = \sum_{k=0}^{N-1} \frac{f^k(x_0)}{k!}(x - x_0)^k = \sum_{k=0}^{N-1} p(k)(x - x_0)^k, \quad (2.61)$$

There are mainly three reasons. First (2.60) is a very close (but not exact) approximation to the very complicated problem of finding the function approximation with a minimum of the maximum error, i.e., an optimization

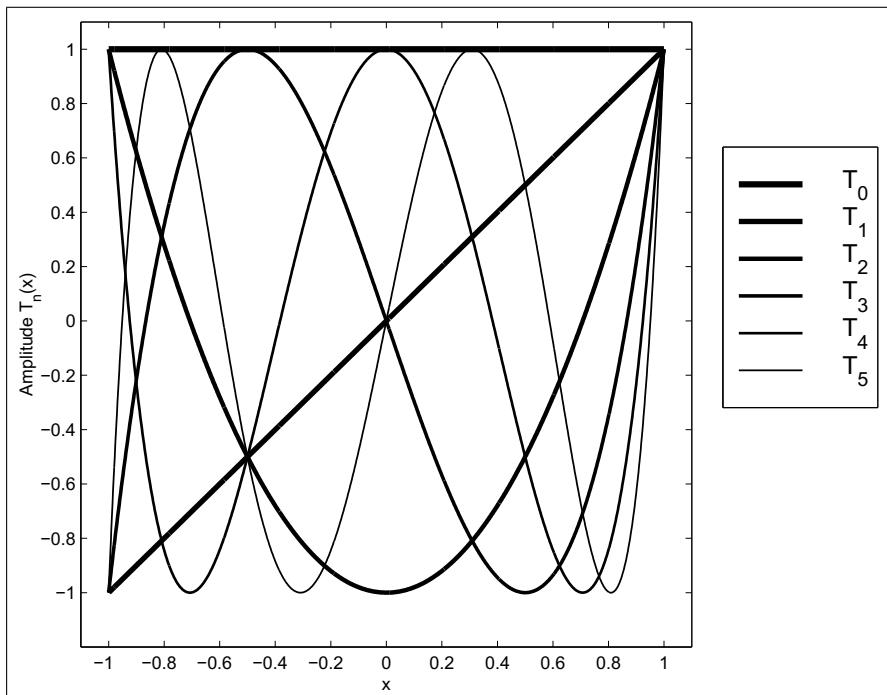


Fig. 2.49. The first six Chebyshev polynomials

of the l_∞ norm $\max(f(x) - \hat{f}(x)) \rightarrow \min$. The second reason we prefer (2.60) is the fact, that a pruned polynomial with $M << N$ still gives a minimum/maximum approximation, i.e., a shorter sum still gives a Chebyshev approximation as if we had started the computation with M as the target from the very start. Last but not least we gain from the fact that (2.60) can be computed (for all functions of relevance) with much fewer coefficients than would be required for a Taylor approximation of the same precision. Let us study these special function approximation in the following for popular functions, like trigonometric, exponential, logarithmic, and the square root functions.

2.10.2 Trigonometric Function Approximation

As a first example we study the inverse tangent function

$$f(x) = \arctan(x), \quad (2.62)$$

where x is specified for the range $-1 \leq x \leq 1$. If we need to evaluate function values outside this interval, we can take advantage of the relation

$$\arctan(x) = 0.5 - \arctan(1/x). \quad (2.63)$$

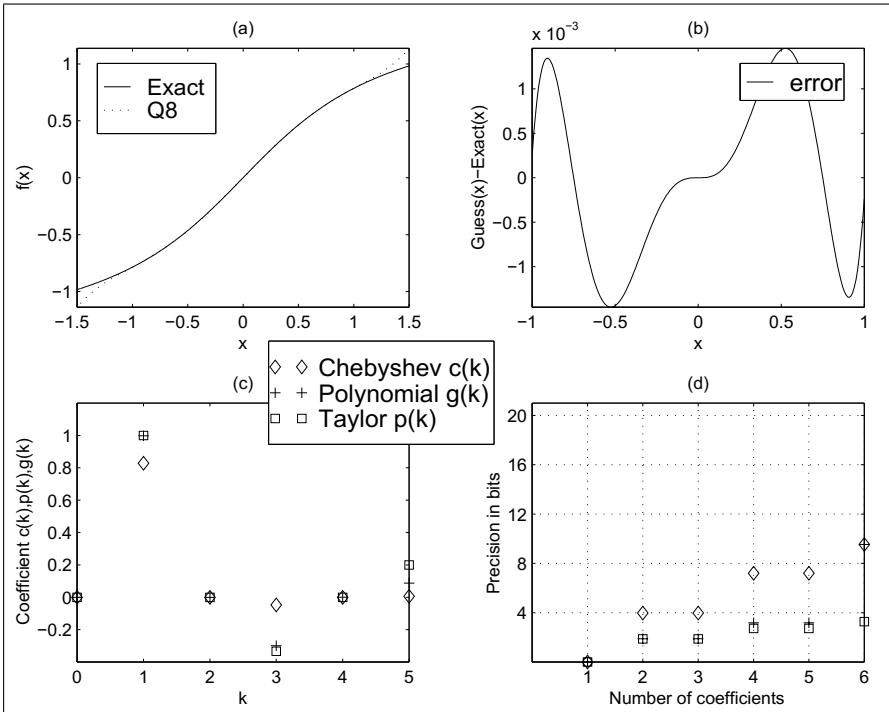


Fig. 2.50. Inverse tangent function approximation. (a) Comparison of full-precision and 8-bit quantized approximations. (b) Error of quantized approximation for $x \in [-1, 1]$. (c) Chebyshev, polynomial from Chebyshev, and Taylor polynomial coefficients. (d) Error of the three pruned polynomials

Embedded multipliers in Altera FPGAs have a basic size of 9×9 bits, i.e., 8 bits plus sign bit data format, or 18×18 bit, i.e., 17 bits plus sign data format. We will therefore in the following always discuss two solutions regarding these two different word sizes.

Fig. 2.50a shows the exact value and approximation for 8-bit quantization, and Fig. 2.50b displays the error, i.e., the difference between the exact function value and the approximation. The error has the typical alternating minimum/maximum behavior of all Chebyshev approximations. The approximation with $N = 6$ already gives an almost perfect approximation. If we use fewer coefficients, e.g., $N = 2$ or $N = 4$, we will have a more-substantial error; see Exercise 2.26 (p. 177).

For 8-bit precision we can see from Fig. 2.50d that $N = 6$ coefficients are sufficient. From Fig. 2.50c we conclude that all even coefficients are zero, because $\arctan(x)$ is an odd symmetric function with respect to $x = 0$. The function to be implemented now becomes

$$\begin{aligned}
 f(x) &= \sum_{k=0}^{N-1} c(k)T_k(x) \\
 f(x) &= c(1)T_1(x) + c(3)T_3(x) + c(5)T_5(x) \\
 f(x) &= 0.8284T_1(x) - 0.0475T_3(x) + 0.0055T_5(x).
 \end{aligned} \tag{2.64}$$

To determine the function values in (2.64) we can substitute the $T_n(x)$ from (2.58) and solve (2.64). It is however more efficient to use the iterative rule (2.59) for the function evaluation. This is known as Clenshaw's recurrence formula [86, p. 193] and works as follows:

$$\begin{aligned}
 d(N) &= d(N+1) = 0 \\
 d(k) &= 2xd(k+1) - d(k+2) + c(k) \quad k = N-1, N-2, \dots, 1 \\
 f(x) &= d(0) = xd(1) - d(2) + c(0)
 \end{aligned} \tag{2.65}$$

For our $N = 6$ system with even coefficients equal to zero we can simplify (2.65) to

$$\begin{aligned}
 d(5) &= c(5) \\
 d(4) &= 2xc(5) \\
 d(3) &= 2xd(4) - d(5) + c(3) \\
 d(2) &= 2xd(3) - d(4) \\
 d(1) &= 2xd(2) - d(3) + c(1) \\
 f(x) &= xd(1) - d(2).
 \end{aligned} \tag{2.66}$$

We can now start to implement the $\arctan(x)$ function approximation in HDL.

Example 2.28: arctan Function Approximation

If we implement the $\arctan(x)$ using the embedded 9×9 bit multipliers we have to take into account that our values are in the range $-1 \leq x < 1$. We therefore use a fractional integer representation in a 1.8 format. In our HDL simulation these fractional numbers are represented as integers and the values are mapped to the range $-256 \leq x < 256$. We can use the same number format for our Chebyshev coefficients because they are all less than 1, i.e., we quantize

$$c(1) = 0.8284 = 212/256, \tag{2.67}$$

$$c(3) = -0.0475 = -12/256, \tag{2.68}$$

$$c(5) = 0.0055 = 1/256. \tag{2.69}$$

The following VHDL code¹⁰ shows the $\arctan(x)$ approximation using polynomial terms up to $N = 6$:

```

PACKAGE n_bits_int IS
    -- User defined types
    SUBTYPE S9 IS INTEGER RANGE -2**8 TO 2**8-1;
    TYPE A1_5S9 IS ARRAY (1 TO 5) OF S9;

```

¹⁰ The equivalent Verilog code `arctan.v` for this example can be found in Appendix A on page 806. Synthesis results are shown in Appendix B on page 881.

```

END n_bits_int;

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-----
ENTITY arctan IS           -----> Interface
  PORT (clk      : IN STD_LOGIC; -- System clock
        reset    : IN STD_LOGIC; -- Asynchronous reset
        x_in     : IN S9;       -- System input
        d_o      : OUT A1_5S9;  -- Auxiliary recurrence
        f_out    : OUT S9);    -- System output
END arctan;
-----
ARCHITECTURE fpga OF arctan IS

  SIGNAL x, f : S9; -- Auxiliary signals
  SIGNAL d : A1_5S9 := (0,0,0,0,0,0); -- Auxiliary array
  -- Chebychev coefficients for 8-bit precision:
  CONSTANT c1 : S9 := 212;
  CONSTANT c3 : S9 := -12;
  CONSTANT c5 : S9 := 1;

BEGIN

  STORE: PROCESS(reset, clk)   -----> I/O store in register
  BEGIN
    IF reset = '1' THEN -- Asynchronous clear
      x <= 0; f_out <= 0;
    ELSIF rising_edge(clk) THEN
      x <= x_in;
      f_out <= f;
    END IF;
  END PROCESS;

  --> Compute sum-of-products:
  SOP: PROCESS(x, d)
  BEGIN
    -- Clenshaw's recurrence formula
    d(5) <= c5;
    d(4) <= x * d(5) / 128;
    d(3) <= x * d(4) / 128 - d(5) + c3;
    d(2) <= x * d(3) / 128 - d(4);
    d(1) <= x * d(2) / 128 - d(3) + c1;
    f <= x * d(1) / 256 - d(2); -- last step is different
  END PROCESS SOP;

  d_o <= d;           -- Provide some test signals as outputs

```

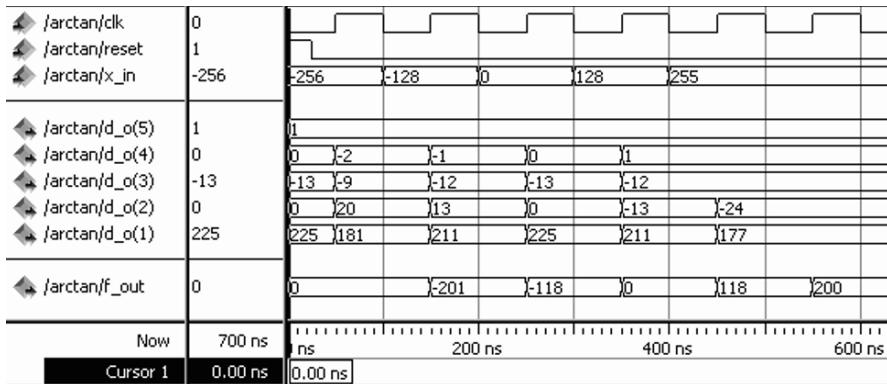


Fig. 2.51. VHDL simulation of the $\arctan(x)$ function approximation for the values $x = -1 = -256/256$, $x = -0.5 = -128/256$, $x = 0$, $x = 0.5 = 128/256$, $x = 1 \approx 255/256$

```
END fpga;
```

The first **PROCESS** is used to infer registers for the input and output data. The next **PROCESS** blocks **SOP** include the computation of the Chebyshev approximation using Clenshaw's recurrence formula. The iteration variables $d(k)$ are also connected to the output ports so we can monitor them. The design uses 106 LEs, three embedded multipliers, and has an **Fmax**=32.71 MHz registered performance using the **TimeQuest** slow 85C model. Comparing FLEX and Cyclone synthesis data we can conclude that the use of embedded multipliers saves many LEs.

A simulation of the **arctan** function approximation is shown in Fig. 2.51. The simulation shows the result for five different input values:

x	$f(x) = \arctan(x)$	$\hat{f}(x)$	$ error $	Eff. bits
-1.0	-0.7854	$-201/256 = -0.7852$	0.0053	7.6
-0.5	-0.4636	$-118/256 = -0.4609$	0.0027	7.4
0	0.0	0	0	—
0.5	0.4636	$118/256 = 0.4609$	0.0027	7.4
1.0	0.7854	$200/256 = 0.7812$	0.0053	7.6

Note that, due to the I/O registers, the output values appear with a delay of one clock cycle.

2.28

If the precision in the previous example is not sufficient we can use more coefficients. The odd Chebyshev coefficients for 16-bit precision, for instance, would be

$$c(2k+1) = (0.82842712, -0.04737854, 0.00487733, -0.00059776, 0.00008001, -0.00001282). \quad (2.70)$$

If we compare this with the Taylor series coefficient

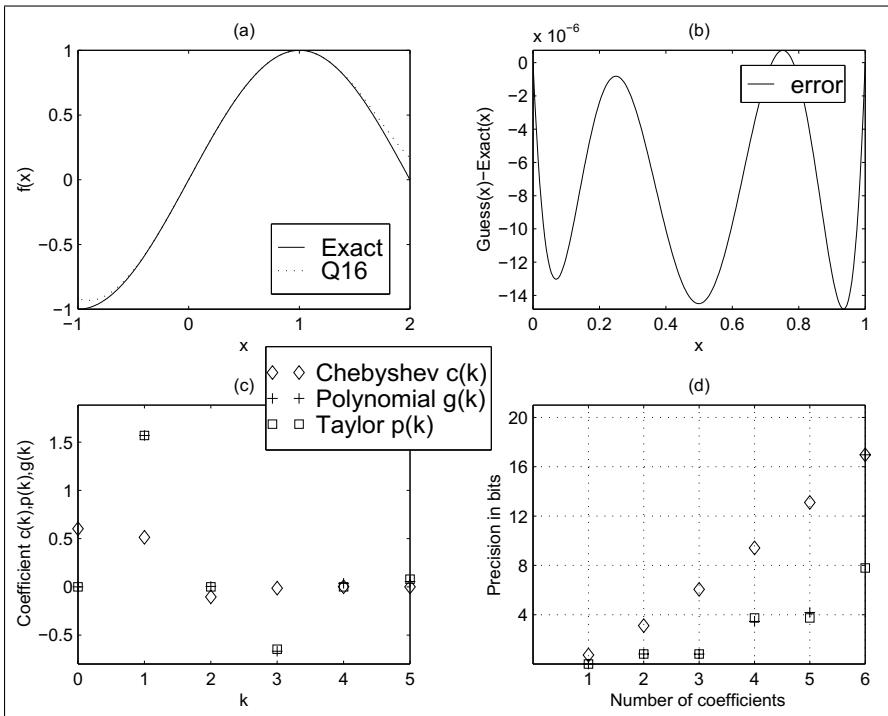


Fig. 2.52. Sine function approximation. (a) Comparison of full-precision and 8-bit quantized approximations. (b) Error of quantized approximation for $x \in [0, 1]$. (c) Chebyshev, polynomial from Chebyshev, and Taylor polynomial coefficients. (d) Error of the three pruned polynomials

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} + \dots (-1)^k \frac{x^{2k+1}}{2k+1} \quad (2.71)$$

$$p(2k+1) = (1, -0.\bar{3}, 0.2, -0.14285714, 0.\bar{1}, -0.\overline{09})$$

we see that the Taylor coefficients converge very slowly compared with the Chebyshev approximation.

There are two more common trigonometric functions. One is the $\sin(x)$ and the other is the $\cos(x)$ function. There is however a small problem with these functions. The argument is usually defined only for the first quadrant, i.e., $0 \leq x \leq \pi/2$, and the other quadrants values are computed via

$$\sin(x) = -\sin(-x) \quad \sin(x) = \sin(\pi/2 - x) \quad (2.72)$$

or equivalent for the $\cos(x)$ we use

$$\cos(x) = \cos(-x) \quad \cos(x) = -\cos(\pi/2 - x). \quad (2.73)$$

We may also find that sometimes the data are normalized $f(x) = \sin(x\pi/2)$ or degree values are used, i.e., $0^\circ \leq x \leq 90^\circ$. Figure 2.52a shows the exact value

and approximation for 16-bit quantization, and Fig. 2.52b displays the error, i.e., the difference between the exact function values and the approximation. In Fig. 2.53 the same data are plotted for the $\cos(x\pi/2)$ function. The problem now is that our Chebyshev polynomials are only defined for the range $x \in [-1, 1]$. Which brings up the question, how the Chebyshev approximation has to be modified to take care of different range values? Luckily this does not take too much effort, we just make a linear transformation of the input values. Suppose the function $f(y)$ to be approximated has a range $y \in [a, b]$ then we can develop our function approximation using a change of variable defined by

$$y = \frac{2x - b - a}{b - a}. \quad (2.74)$$

Now if we have for instance in our $\sin(x\pi/2)$ function x in the range $x = [0, 1]$, i.e., $a = 0$ and $b = 1$, it follows that y has the range $y = [(2 \times 0 - 1 - 0)/(1 - 0), (2 \times 1 - 1 - 0)/(1 - 0)] = [-1, 1]$, which we need for our Chebyshev approximation. If we prefer the degree representation then $a = 0$ and $b = 90$, and we will use the mapping $y = (2x - 90)/90$ and develop the Chebyshev approximation in y .

The final question we discuss is regarding the polynomial computation. You may ask if we really need to compute the Chebyshev approximation via the Clenshaw's recurrence formula (2.65) or if we can use instead the direct polynomial approximation, which requires one fewer add operation per iteration:

$$f(x) = \sum_{k=0}^{N-1} p(k)x^k \quad (2.75)$$

or even better use the Horner scheme

$$\begin{aligned} s(N-1) &= p(N-1) \\ s(k) &= s(k+1) \times x + p(k) \quad k = N-2, \dots, 0. \\ f &= s(0). \end{aligned} \quad (2.76)$$

We can of course substitute the Chebyshev functions (2.58) in the approximation formula (2.60), because the $T_n(x)$ do not have terms of higher order than x^n . However there is one important disadvantage to this approach. We will lose the pruning property of the Chebyshev approximation, i.e., if we use in the polynomial approximation (2.75) fewer than N terms, the pruned polynomial will no longer be an l_∞ optimized polynomial. Figure 2.50d (p. 144) shows this property. If we use all 6 terms the Chebyshev and the associated polynomial approximation will have the same precision. If we now prune the polynomial, the Chebyshev function approximation (2.60) using the $T_n(x)$ has more precision than the pruned polynomial using (2.75). The resulting precision is much lower than the equivalent pruned Chebyshev function approximation of the same length. In fact it is not much better than the Taylor

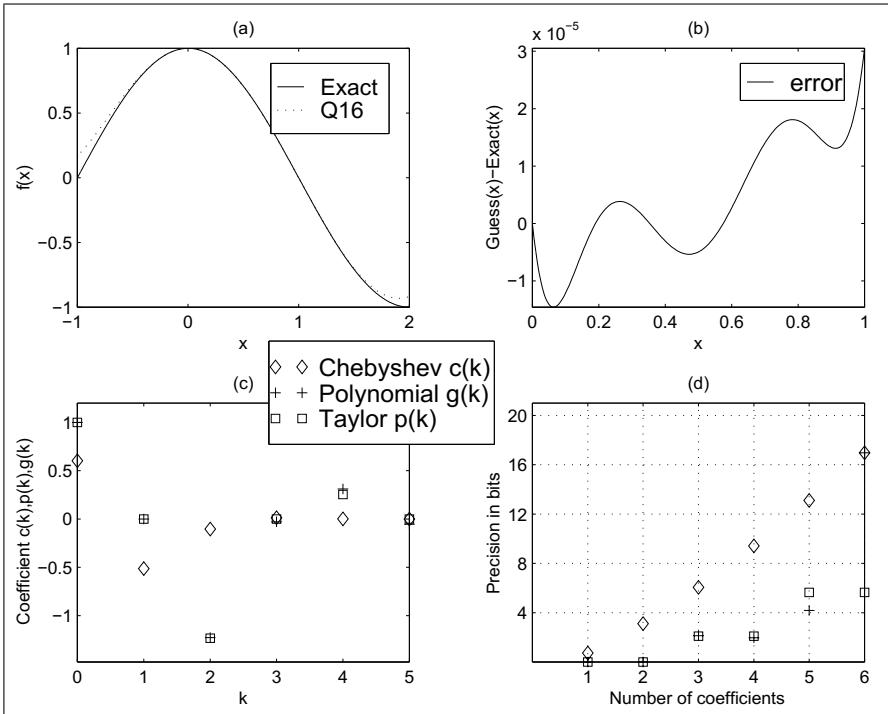


Fig. 2.53. Cosine function approximation. (a) Comparison of full-precision and 16-bit quantized approximations. (b) Error of quantized approximation for $x \in [0, 1]$. (c) Chebyshev, polynomial from Chebyshev, and Taylor polynomial coefficients. (d) Error of the three pruned polynomials

approximation. So the solution to this problem is not complicated: if we want to shorten the length $M < N$ of our polynomial approximation (2.75) we need to develop first a Chebyshev approximation for length M and then compute the polynomial coefficient $g(k)$ from this pruned Chebyshev approximation. Let us demonstrate this with a comparison of 8- and 16-bit $\arctan(x)$ coefficients. The substitution of the Chebyshev functions (2.58) into the coefficient (2.70) gives the following odd coefficients:

$$\begin{aligned} g(2k+1) = & (0.99999483, -0.33295711, 0.19534659, \\ & -0.12044859, 0.05658999, -0.01313038). \end{aligned} \quad (2.77)$$

If we now use the length $N = 6$ approximation from (2.67) the odd coefficient will be

$$g(2k+1) = (0.9982, -0.2993, 0.0876). \quad (2.78)$$

Although the pruned Chebyshev coefficients are the same, we see from a comparison of (2.77) and (2.78) that the polynomial coefficient differ essentially. The coefficient $g(5)$ for instance has a factor of 2 difference.

We can summarize the Chebyshev approximation in the following procedure.

Algorithm 2.29: Chebyshev Function Approximation

- 1) Define the number of coefficients N .
- 2) Transform the variable from x to y using (2.74)
- 3) Determine the Chebyshev approximation in y .
- 4) Determine the direct polynomial coefficients $g(k)$ using Clenshaw's recurrence formula.
- 5) Build the inverse of the mapping y .

If we apply these five steps to our $\sin(x\pi/2)$ function for $x \in [0, 1]$ with four nonzero coefficients, we get the following polynomials sufficient for a 16-bit quantization

$$\begin{aligned} f(x) &= \sin(x\pi/2) \\ &= 1.57035062x + 0.00508719x^2 - 0.66666099x^3 \\ &\quad + 0.03610310x^4 + 0.05512166x^5 \\ &= (51457x + 167x^2 - 21845x^3 + 1183x^4 + 1806x^5)/32768. \end{aligned}$$

Note that the first coefficient is larger than 1 and we need to scale appropriate. This is quite different from the Taylor approximation given by

$$\begin{aligned} \sin\left(\frac{x\pi}{2}\right) &= \frac{x\pi}{2} - \frac{1}{3!}\left(\frac{x\pi}{2}\right)^3 + \frac{1}{5!}\left(\frac{x\pi}{2}\right)^5 \\ &\quad + \dots + \frac{(-1)^k}{(2k+1)!}\left(\frac{x\pi}{2}\right)^{2k+1}. \end{aligned}$$

Figure 2.52c shows a graphical illustration. For an 8-bit quantization we would use

$$\begin{aligned} f(x) = \sin(x\pi/2) &= 1.5647x + 0.0493x^2 - 0.7890x^3 + 0.1748x^4 \\ &= (200x + 6x^2 - 101x^3 + 22x^4)/128. \end{aligned} \quad (2.79)$$

Although we would expect that, for an odd symmetric function, all even coefficients are zero, this is not the case in this approximation, because we only used the interval $x \in [0, 1]$ for the approximation. The $\cos(x)$ function can be derived via the relation

$$\cos\left(\frac{x\pi}{2}\right) = \sin\left((x+1)\frac{\pi}{2}\right) \quad (2.80)$$

or we may also develop a direct Chebyshev approximation. For $x \in [0, 1]$ with four nonzero coefficients and get the following polynomial for a 16-bit quantization

$$\begin{aligned} f(x) &= \cos\left(\frac{x\pi}{2}\right) \\ &= 1.00000780 - 0.00056273x - 1.22706059x^2 \\ &\quad - 0.02896799x^3 + 0.31171138x^4 - 0.05512166x^5 \\ &= (32768 - 18x - 40208x^2 - 949x^3 + 10214x^4 - 1806x^5)/32768. \end{aligned}$$

For an 8-bit quantization we would use

$$f(x) = \cos\left(\frac{x\pi}{2}\right) \\ = (0.9999 + 0.0046x - 1.2690x^2 + 0.0898x^3 + 0.1748x^4) \quad (2.81)$$

$$= (128 + x - 162x^2 + 11x^3 + 22x^4)/128. \quad (2.82)$$

Again the Taylor approximation has quite different coefficients:

$$\cos\left(\frac{x\pi}{2}\right) = 1 - \frac{1}{2!}\left(\frac{x\pi}{2}\right)^2 + \frac{1}{4!}\left(\frac{x\pi}{2}\right)^4 + \dots + \frac{(-1)^k}{(2k)!}\left(\frac{x\pi}{2}\right)^{2k}.$$

Figure 2.52c shows a graphical illustration of the coefficients. We notice from Fig. 2.52d that with the same number (i.e., six) of terms x^k the Taylor approximation only provides about 6 bit accuracy, while the Chebyshev approximation has 16-bit precision.

2.10.3 Exponential and Logarithmic Function Approximation

The exponential function is one of the few functions who's Taylor approximation converges relatively fast. The Taylor approximation is given by

$$f(x) = e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \dots + \frac{x^k}{k!} \quad (2.83)$$

with $0 \leq x \leq 1$. For 16-bit polynomial quantization computed using the Chebyshev coefficients we would use:

$$f(x) = e^x \\ = 1.00002494 + 0.99875705x + 0.50977984x^2 \\ + 0.14027504x^3 + 0.06941551x^4 \\ = (32769 + 32727x + 16704x^2 + 4597x^3 + 2275x^4)/32768.$$

Only terms up to order x^4 are required to reach 16-bit precision. We notice also from Fig. 2.54c that the Taylor and polynomial coefficient computed from the Chebyshev approximation are quite similar. If 8 bits plus sign precision are sufficient, we use

$$f(x) = e^x = 1.0077 + 0.8634x + 0.8373x^2 \\ = (129 + 111x + 107x^2)/128. \quad (2.84)$$

Based on the fact that one coefficient is larger than $c(0) > 1.0$ we need to select a scaling factor of 128.

The input needs to be scaled in such a way that $0 \leq x \leq 1$. If x is outside this range we can use the identity

$$e^{sx} = (e^x)^s \quad (2.85)$$

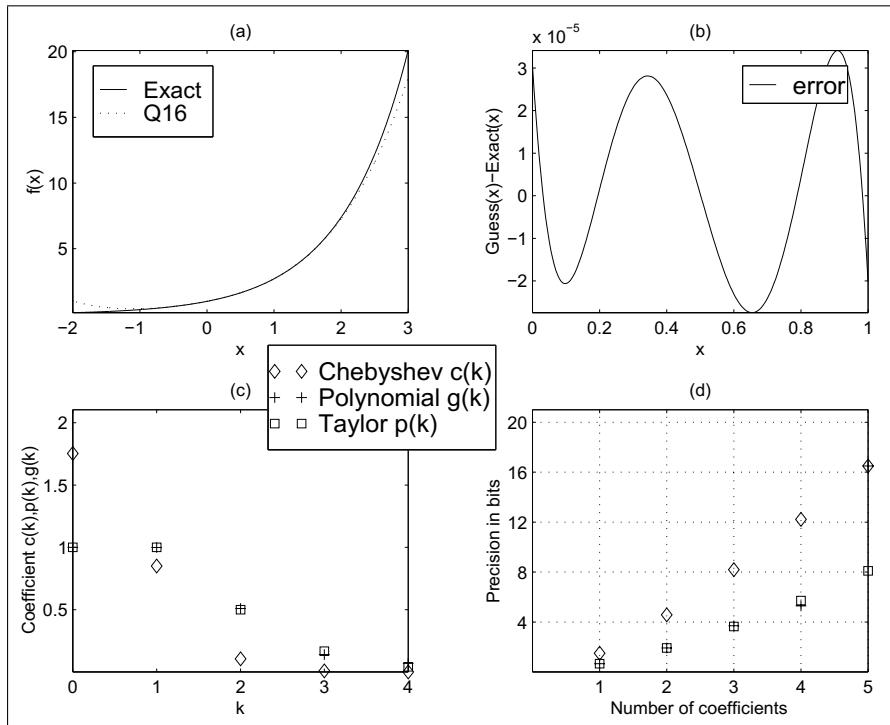


Fig. 2.54. Exponential $f(x) = \exp(x)$ function approximation. (a) Comparison of full-precision and 16-bit quantized approximations. (b) Error of quantized approximation for $x \in [0, 1]$. (c) Chebyshev, polynomial from Chebyshev, and Taylor polynomial coefficients. (d) Error of the three pruned polynomials

Because $s = 2^k$ is a power-of-two value this implies that a series of squaring operations need to follow the exponential computation. For a negative exponent we can use the relation

$$e^{-x} = \frac{1}{e^x}, \quad (2.86)$$

or develop a separate approximation. If we like to build a direct function approximation to $f(x) = e^{-x}$ we have to alternate the sign of each second term in (2.83). For a Chebyshev polynomial approximation we get additional minor changes in the coefficients. For a 16-bit Chebyshev polynomial approximation we use

$$\begin{aligned} f(x) &= e^{-x} \\ &= 0.99998916 - 0.99945630x + 0.49556967x^2 \\ &\quad - 0.15375046x^3 + 0.02553654x^4 \\ &= (65535 - 65500x + 32478x^2 - 10076x^3 + 1674x^4)/65536. \end{aligned}$$

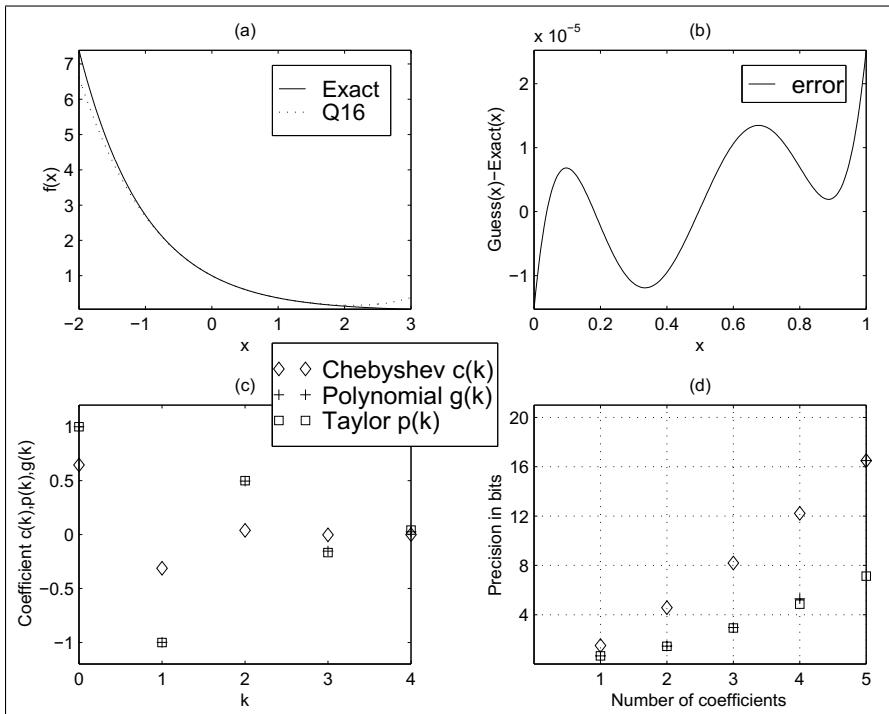


Fig. 2.55. Negative exponential $f(x) = \exp(-x)$ function approximation. (a) Comparison of full-precision and 16-bit quantized approximations. (b) Error of quantized approximation for $x \in [0, 1]$. (c) Chebyshev, polynomial from Chebyshev, and Taylor polynomial coefficients. (d) Error of the three pruned polynomials

where x is defined for the range $x \in [0, 1]$. Note that, based on the fact that all coefficients are less than 1, we can select a scaling by a factor of 2 larger than in (2.84). From Fig. 2.55d we conclude that three or five coefficients are required for 8- and 16-bit precision, respectively. For 8-bit quantization we would use the coefficients

$$\begin{aligned} f(x) &= e^{-x} = 0.9964 - 0.9337x + 0.3080x^2 \\ &= (255 - 239x + 79x^2)/256. \end{aligned} \quad (2.87)$$

The inverse to the exponential function is the logarithm function, which is typically approximated for the argument in the range $[1, 2]$. As notation this is typically written as $f(x) = \ln(1 + x)$ now with $0 \leq x \leq 1$. Figure 2.56a shows the exact and 16-bit quantized approximation for this range. The approximation with $N = 6$ gives an almost perfect approximation. If we use fewer coefficients, e.g., $N = 2$ or $N = 3$, we will have a more substantial error; see Exercise 2.29 (p. 177).

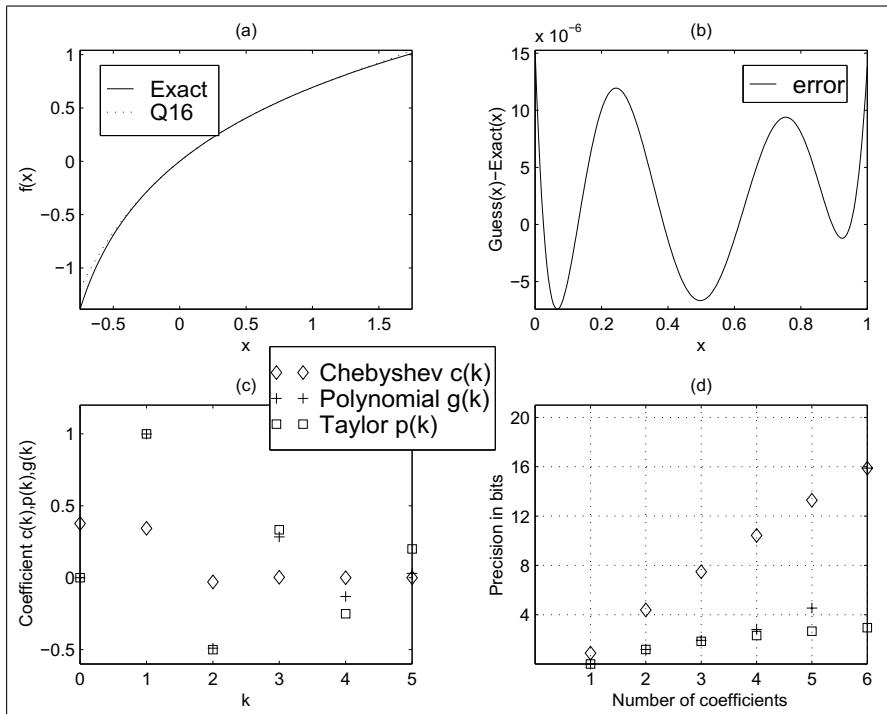


Fig. 2.56. Natural logarithm $f(x) = \ln(1+x)$ function approximation. (a) Comparison of full-precision and 16-bit quantized approximations. (b) Error of quantized approximation for $x \in [0, 1]$. (c) Chebyshev, polynomial from Chebyshev, and Taylor polynomial coefficients. (d) Error of the three pruned polynomials

The Taylor series approximation is no longer fast converging as for the exponential function

$$f(x) = \ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} + \dots + \frac{(-1)^{k+1}x^k}{k}$$

as can be seen from the linear factor in the denominator. A 16-bit Chebyshev approximation converges much faster, as can be seen from Fig. 2.56d. Only six coefficients are required for 16-bit precision. With six Taylor coefficients we get less than 4-bit precision. For 16-bit polynomial quantization computed using the Chebyshev coefficients we would use

$$\begin{aligned} f(x) &= \ln(1+x) \\ &= 0.00001145 + 0.99916640x - 0.48969909x^2 \\ &\quad + 0.28382318x^3 - 0.12995720x^4 + 0.02980877x^5 \\ &= (1 + 65481x - 32093x^2 + 18601x^3 - 8517x^4 + 1954x^5)/65536. \end{aligned}$$

Only terms up to order x^5 are required to get 16-bit precision. We also notice from Fig. 2.56c that the Taylor and polynomial coefficient computed from the Chebyshev approximation are similar only for the first three coefficients.

We can now start to implement the $\ln(1 + x)$ function approximation in HDL.

Example 2.30: $\ln(1+x)$ Function Approximation

If we implement the $\ln(1 + x)$ using embedded 18×18 bit multipliers we have to take into account that our values x are in the range $0 \leq x < 1$. We therefore use a fractional integer representation with a 2.16 format. We use an additional guard bit that guarantees no problem with any overflow and that $x = 1$ can be exactly represented as 2^{16} . We use the same number format for our Chebyshev coefficients because they are all less than 1.

The following VHDL code¹¹ shows the $\ln(1 + x)$ approximation using six coefficients:

```

PACKAGE n_bits_int IS           -- User defined types
  SUBTYPE S9 IS INTEGER RANGE -2**8 TO 2**8-1;
  SUBTYPE S18 IS INTEGER RANGE -2**17 TO 2**17-1;
  TYPE A0_5S18 IS ARRAY (0 TO 5) of S18;
END n_bits_int;

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-----
ENTITY ln IS                   -----> Interface
  GENERIC (N : INTEGER := 5); -- Number of Coeffcients-1
  PORT (clk      : IN STD_LOGIC; -- System clock
        reset    : IN STD_LOGIC; -- Asynchron reset
        x_in     : IN S18;       -- System input
        f_out    : OUT S18:=0);  -- System output
END ln;
-----
ARCHITECTURE fpga OF ln IS

  SIGNAL x, f : S18:= 0; -- Auxilary wire
-- Polynomial coefficients for 16 bit precision:
-- f(x) = (1 + 65481 x -32093 x^2 + 18601 x^3
--          -8517 x^4 + 1954 x^5)/65536
  CONSTANT p : A0_5S18 :=
    (1,65481,-32093,18601,-8517,1954);
  SIGNAL s : A0_5S18;

BEGIN

  STORE: PROCESS(reset, clk)   -----> I/O store in register

```

¹¹ The equivalent Verilog code `ln.v` for this example can be found in Appendix A on page 807. Synthesis results are shown in Appendix B on page 881.

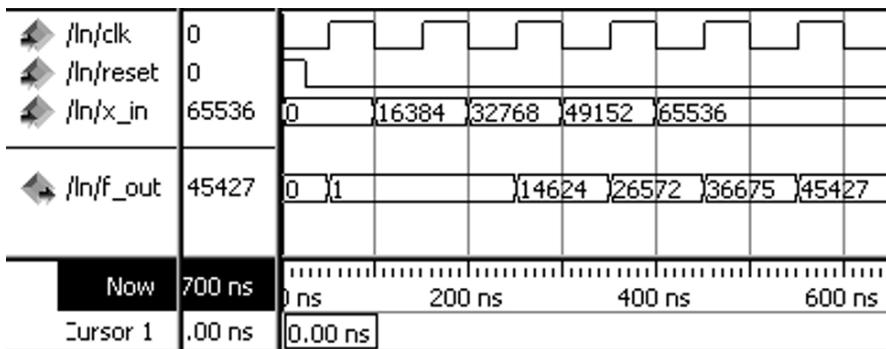


Fig. 2.57. VHDL simulation of the $\ln(1 + x)$ function approximation for the values $x = 0, x = 0.25 = 16384/65536, x = 0.5 = 32768/65536, x = 0.75 = 49152/65536, x = 1.0 = 65536/65536$

```

BEGIN
  IF reset = '1' THEN -- Asynchronous clear
    x <= 0; f_out <= 0;
  ELSIF rising_edge(clk) THEN
    x <= x_in;
    f_out <= f;
  END IF;
END PROCESS;

--> Compute sum-of-products:
SOP: PROCESS (x,s)
VARIABLE slv : STD_LOGIC_VECTOR(35 DOWNTO 0);
BEGIN
  -- Polynomial Approximation from Chebyshev coefficients
  s(N) <= p(N);
  FOR K IN N-1 DOWNTO 0 LOOP
    slv := CONV_STD_LOGIC_VECTOR(x,18)
      * CONV_STD_LOGIC_VECTOR(s(K+1),18);
    s(K) <= CONV_INTEGER(slv(33 downto 16)) + p(K);
  END LOOP;          -- x*s/65536 problem 32 bits
  f <= s(0);         -- make visible outside
END PROCESS SOP;

END fpga;

```

The first PROCESS is used to infer the registers for the input and output data. The next PROCESS blocks SOP includes the computation of the Chebyshev approximation using a sum of product computations. The multiply and scale arithmetic is implemented with standard logic vectors data types because the 36-bit products are larger than the valid 32-bit range allowed for integers. The design uses 88 LEs, ten embedded 9×9 -bit multipliers (or half of that for 18×18 -bit multipliers), and has an Fmax=29.2 MHz registered performance using the TimeQuest slow 85C model.

A simulation of the function approximation is shown in Fig. 2.57. The simulation shows the result for five different input values:

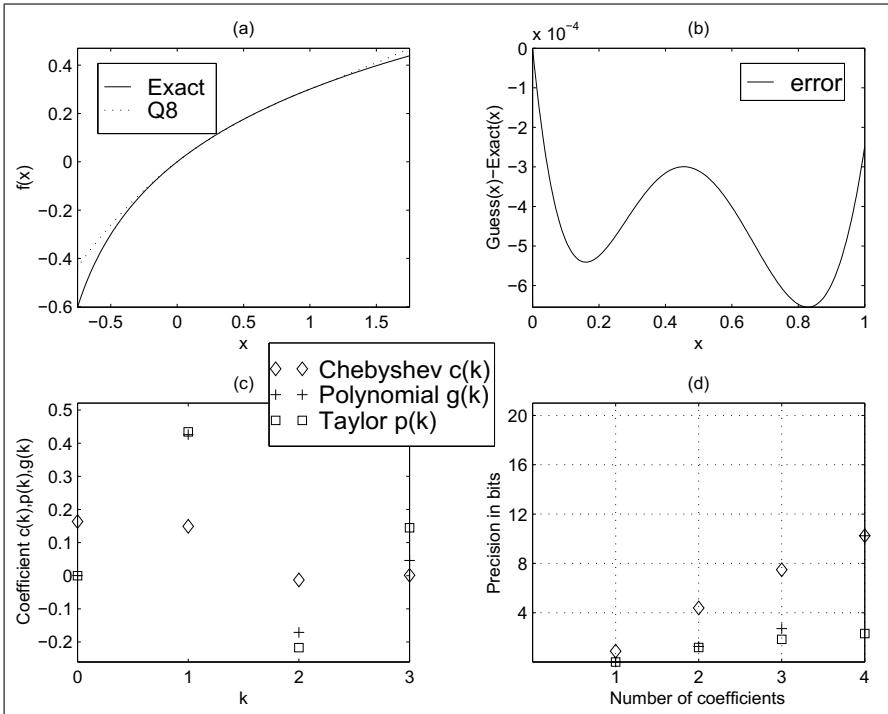


Fig. 2.58. Base 10 logarithm $f(x) = \log_{10}(1+x)$ function approximation. (a) Comparison of full-precision and 8-bit quantized approximations. (b) Error of quantized approximation for $x \in [0, 1]$. (c) Chebyshev, polynomial from Chebyshev, and Taylor polynomial coefficients. (d) Error of the three pruned polynomials

x	$f(x) = \ln(x)$	$\hat{f}(x)$	$ \text{error} $	Eff. bits
0	0	1	1.52×10^{-5}	16
0.25	$14623.9/2^{16}$	$14624/2^{16}$	4.39×10^6	17.8
0.5	$26572.6/2^{16}$	$26572/2^{16}$	2.11×10^5	15.3
0.75	$36675.0/2^{16}$	$36675/2^{16}$	5.38×10^7	20.8
1.0	$45426.1/2^{16}$	$45427/2^{16}$	1.99×10^5	15.6

Note that, due to the I/O registers, the output values appear with a delay of one clock cycle.

2.30

If we compare the polynomial code of the \ln function with Clenshaw's recurrence formula from Example 2.28 (p. 145), we notice the reduction by one adder in the design.

If 8 bit plus sign precision is sufficient, we use

$$\begin{aligned} f(x) = \ln(1 + x) &= 0.0006 + 0.9813x - 0.3942x^2 + 0.1058x^3 \\ &= (251x - 101x^2 + 27x^3)/256. \end{aligned} \quad (2.88)$$

Based on the fact that no coefficient is larger than 1.0 we can select a scaling factor of 256.

If the argument x is not in the valid range $[0, 1]$, using the following algebraic manipulation with $y = sx = 2^k x$ we get

$$\ln(sx) = \ln(s) + \ln(x) = k \times \ln(2) + \ln(x), \quad (2.89)$$

i.e., we normalize by a power-of-two factor such that x is again in the valid range. If we have determined s , the addition arithmetic effort is only one multiply and one add operation.

If we like to change to another base, e.g., base 10, we can use the following rule:

$$\log_a(x) = \ln(x) / \ln(a), \quad (2.90)$$

i.e., we only need to implement the logarithmic function for one base and can deduce it for any other base. On the other hand the divide operation may be expensive to implement too and we can alternatively develop a separate Chebyshev approximation. For base 10 we would use, in 16-bit precision, the following Chebyshev polynomial coefficients:

$$\begin{aligned} f(x) &= \log_{10}(1+x) \\ &= 0.00000497 + 0.43393245x - 0.21267361x^2 \\ &\quad + 0.12326284x^3 - 0.05643969x^4 + 0.01294578x^5 \\ &= (28438x - 13938x^2 + 8078x^3 - 3699x^4 + 848x^5) / 65536 \end{aligned}$$

for $x \in [0, 1]$. Figure 2.58a shows the exact and 8-bit quantized function of $\log_{10}(1+x)$. For an 8-bit quantization we would use the following approximation:

$$\begin{aligned} f(x) &= \log_{10}(1+x) \\ &= 0.0002 + 0.4262x - 0.1712x^2 + 0.0460x^3 \end{aligned} \quad (2.91)$$

$$= (109x - 44x^2 + 12x^3) / 256, \quad (2.92)$$

which uses only three nonzero coefficients, as shown in Fig. 2.58d.

2.10.4 Square Root Function Approximation

The development of a Taylor function approximation for the square root cannot be computed around $x_0 = 0$ because then all derivatives $f^n(x_0)$ would be zero or even worse 1/0. However, we can compute a Taylor series around $x_0 = 1$ for instance. The Taylor approximation would then be

$$\begin{aligned} f(x) &= \sqrt{x} \\ &= \frac{(x-1)^0}{0!} + 0.5 \frac{(x-1)^1}{1!} - \frac{0.5^2}{2!} (x-1)^2 + \frac{0.5^2 \cdot 1.5}{3!} (x-1)^3 - \dots \\ &= 1 + \frac{x-1}{2} - \frac{(x-1)^2}{8} + \frac{(x-1)^3}{16} - \frac{5}{128} (x-1)^4 + \dots \end{aligned}$$

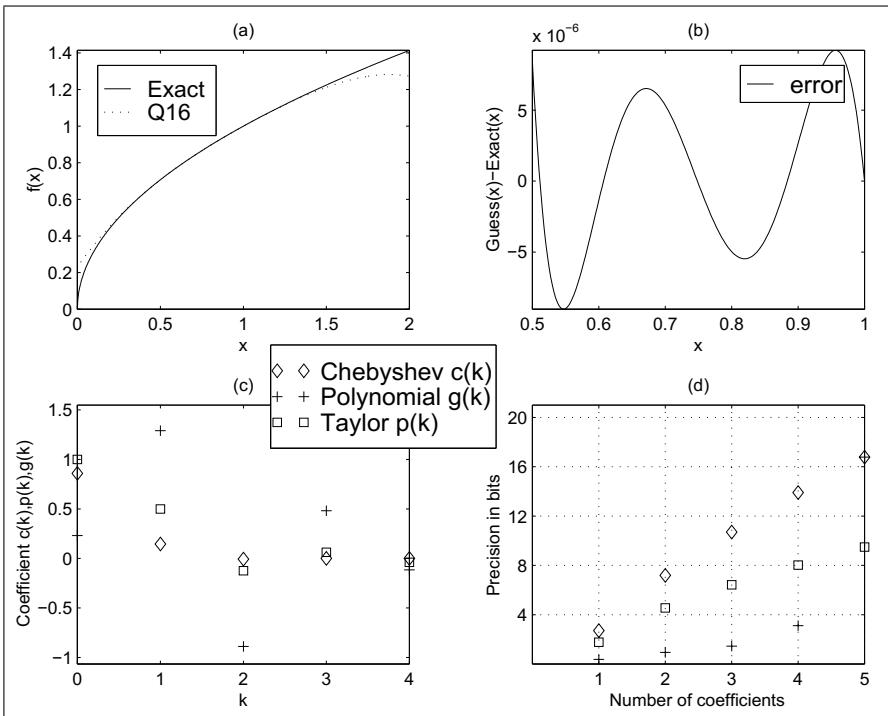


Fig. 2.59. Square root $f(x) = \sqrt{x}$ function approximation. (a) Comparison of full-precision and 16-bit quantized approximations. (b) Error of quantized approximation for $x \in [0.5, 1]$. (c) Chebyshev, polynomial from Chebyshev, and Taylor polynomial coefficients. (d) Error of the three pruned polynomials

The coefficient and the equivalent Chebyshev coefficient are graphically interpreted in Fig. 2.59c. For 16-bit polynomial quantization computed using the Chebyshev coefficient we would use

$$\begin{aligned}
 f(x) &= \sqrt{x} \\
 &= 0.23080201 + 1.29086721x - 0.88893983x^2 \\
 &\quad + 0.48257525x^3 - 0.11530993x^4 \\
 &= (7563 + 42299x - 29129x^2 + 15813x^3 - 3778x^4)/32768.
 \end{aligned}$$

The valid argument range is $x \in [0.5, 1]$. Only terms up to order x^4 are required to get 16-bit precision. We also notice from Fig. 2.59c that the Taylor and polynomial coefficients computed from the Chebyshev approximation are not similar. The approximation with $N = 5$ shown in Fig. 2.59a is almost a perfect approximation. If we use fewer coefficients, e.g., $N = 2$ or $N = 3$, we will have a more-substantial error; see Exercise 2.30 (p. 178).

The only thing left to discuss is the question of how to handle argument values outside the range $0.5 \leq x < 1$. For the square root operation this can

be done by splitting the argument $y = sx$ into a power-of-two scaling factor $s = 2^k$ and the remaining argument with a valid range of $0.5 \leq x < 1$. The square root for the scaling factor is accomplished by

$$\sqrt{s} = \sqrt{2^k} = \begin{cases} 2^{k/2} & k \text{ even} \\ \sqrt{2} \times 2^{(k-1)/2} & k \text{ odd} \end{cases} \quad (2.93)$$

We can now start to implement the \sqrt{x} function approximation in HDL.

Example 2.31: Square Root Function Approximation

We can implement the function approximation in a parallel way using N embedded 18×18 bit multiplier or we can build an FSM to solve this iteratively. Other FSM design examples can be found in Exercises 2.20 , p. 172 and 2.21, p. 173. In a first design step we need to scale our data and coefficients in such a way that overflow-free processing is guaranteed. In addition we need a pre- and post-scaling such that x is in the range $0.5 \leq x < 1$. We therefore use a fractional integer representation in 3.15 format. We use two additional guard bits that guarantee no problem with any overflow and that $x = 1$ can be exactly represented as 2^{15} . We use the same number format for our Chebyshev coefficients because they are all less than 2.

The following VHDL code¹² shows the \sqrt{x} approximation using $N = 5$ coefficients:

```

PACKAGE n_bits_int IS           -- User defined types
  SUBTYPE S9 IS INTEGER RANGE -2**8 TO 2**8-1;
  SUBTYPE S17 IS INTEGER RANGE -2**16 TO 2**16-1;
  TYPE A0_4S17 IS ARRAY (0 TO 4) OF S17;
  TYPE STATE_TYPE IS
    (start, leftshift, sop, rightshift, done);
  TYPE OP_TYPE IS (load, mac, scale, denorm, nop);
END n_bits_int;

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-----

ENTITY sqrt IS           -----> Interface
  PORT (clk      : IN STD_LOGIC; -- System clock
        reset    : IN STD_LOGIC; -- Asynchronous reset
        count_o  : OUT INTEGER RANGE 0 TO 3; -- Counter SLL
        x_in     : IN S17; -- System input
        pre_o    : OUT S17; -- Prescaler
        x_o      : OUT S17; -- Normalized x_in
        post_o   : OUT S17; -- Postscaler
        ind_o    : OUT INTEGER RANGE -1 TO 4; -- Index to p
        imm_o    : OUT S17; -- ALU preload value
        a_o      : OUT S17; -- ALU factor
      );

```

¹² The equivalent Verilog code `sqrt.v` for this example can be found in Appendix A on page 809. Synthesis results are shown in Appendix B on page 881.

```

        f_o      : OUT S17; -- ALU output
        f_out    : OUT S17); -- System output
END sqrt;
-----
ARCHITECTURE fpga OF sqrt IS

    SIGNAL s      : STATE_TYPE;
    SIGNAL op     : OP_TYPE;

    SIGNAL x : S17 := 0; -- Auxilary
    SIGNAL a, b, f, imm : S17 := 0; -- ALU data
    -- Chebychev poly coefficients for 16 bit precision:
    CONSTANT p : A0_4S17 :=
        (7563,42299,-29129,15813,-3778);
    SIGNAL pre, post : S17;

BEGIN

    States: PROCESS(clk, reset) -----> SQRT in behavioral style
    VARIABLE ind : INTEGER RANGE -1 TO 4 := 0;
    VARIABLE count : INTEGER RANGE 0 TO 3;
    BEGIN
        IF reset = '1' THEN                      -- Asynchronous reset
            s <= start;
            f_out <= 0;
        ELSIF rising_edge(clk) THEN
            CASE s IS                         -- Next State assignments
                WHEN start =>                 -- Initialization step
                    s <= leftshift;
                    ind := 4;
                    imm <= x_in;           -- Load argument in ALU
                    op <= load;
                    count := 0;
                WHEN leftshift =>             -- Normalize to 0.5 .. 1.0
                    count := count + 1;
                    a <= pre;
                    op <= scale;
                    imm <= p(4);
                IF count = 2 THEN
                    op <= NOP;
                END IF;
                IF count = 3 THEN -- Normalize ready ?
                    s <= sop;
                    op <= load;
                    x <= f;
                END IF;
                WHEN sop =>                  -- Processing step
                    ind := ind - 1;
                    a <= x;
                IF ind =-1 THEN -- SOP ready ?
                    s <= rightshift;
                    op <= denorm;
                    a <= post;
                END IF;
            END CASE;
        END IF;
    END PROCESS;
END fpga;

```

```

    ELSE
        imm <= p(ind);
        op <= mac;
    END IF;
WHEN rightshift => -- Denormalize to original range
    s <= done;
    op <= nop;
WHEN done => -- Output of results
    f_out <= f;      -----> I/O store in register
    op <= nop;
    s <= start;      -- start next cycle
END CASE;
END IF;
ind_o <= ind;
count_o <= count;
END PROCESS States;

ALU: PROCESS(clk, reset)
BEGIN
    IF reset = '1' THEN           -- Asynchronous clear
        f <= 0;
    ELSIF rising_edge(clk) THEN
        CASE OP IS
            WHEN load => f <= imm;
            WHEN mac => f <= a * f /32768 + imm;
            WHEN scale => f <= a * f;
            WHEN denorm => f <= a * f /32768;
            WHEN nop => f <= f;
            WHEN others => f <= f;
        END CASE;
    END IF;
END PROCESS ALU;

EXP: PROCESS(x_in)
VARIABLE slv : STD_LOGIC_VECTOR(16 DOWNTO 0);
VARIABLE po, pr : S17;
BEGIN
    slv := CONV_STD_LOGIC_VECTOR(x_in, 17);
    pr := 2**14;      -- Compute pre- and post scaling
    pre <= 0;
    FOR K IN 0 TO 15 LOOP
        IF slv(K) = '1' THEN
            pre <= pr;
        END IF;
        pr := pr / 2;
    END LOOP;
    po := 1;          -- Compute pre- and post scaling
    FOR K IN 0 TO 7 LOOP
        IF slv(2*K) = '1' THEN -- even 2^k get 2^k/2
            po := 256 * 2**K;
        END IF;
    -- sqrt(2): CSD Error = 0.0000208 = 15.55 effective bits
    -- +1 +0. -1 +0 -1 +0 +1 +0 +0 +0 +0 +1

```

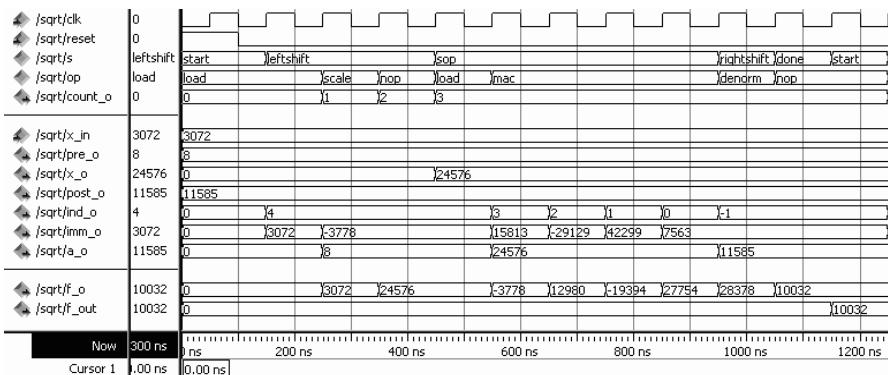


Fig. 2.60. VHDL simulation of the \sqrt{x} function approximation for the value $x = 0.75/8 = 3072/32768$

```
-- 9      7      5      3      1      -5
      IF slv(2*K+1) = '1' THEN -- odd k has sqrt(2) factor
          po := 2** (K+9)-2** (K+7)-2** (K+5)+2** (K+3)
                           +2** (K+1)+2** K/32;
      END IF;
      post <= po;
END LOOP;

END PROCESS EXP;

a_o <= a; -- Provide some test signals as outputs
imm_o <= imm;
f_o <= f;
pre_o <= pre;
post_o <= post;
x_o <= x;

END fpga;
```

The code consists of three major PROCESS blocks. The control part is placed in the FSM block, while the arithmetic parts can be found in the ALU and EXP blocks. The first FSM PROCESS is used to control the machine and place the data in the correct registers for the ALU and EXP blocks. In the **start** state the data are initialized and the input data are loaded into the ALU. In the **leftshift** state the input data are normalized such that the input x is in the range $x \in [0.5, 1)$. The **sop** state is the major processing step where the polynomial evaluation takes place using multiply-accumulate operations performed by the ALU. At the end data are loaded for the denormalization step, i.e., **rightshift** state, that reverses the normalization done before. In the final step the result is transferred to the output register and the FSM is ready for the next square root computation. The ALU PROCESS block performs a $f = a \times f + \text{imm}$ operation as used in the Horner scheme (2.76), p. 149 to compute the polynomial function and will be synthesized to a single 18×18 embedded multiplier (or two 9×9 -bit multiplier blocks as reported by Quartus) and some additional add and normalization logic. The block has the form of an ALU, i.e., the signal **op** is used to determine the current operation.

The accumulator register **f** can be preloaded with an **imm** operand. The last PROCESS block EXP hosts the computation of the pre- and post-normalization factors according to (2.93). The $\sqrt{2}$ factor for the odd k values of 2^k has been implemented with CSD code computed with the **csd.exe** program. The design uses 261 LEs, two embedded 9×9 -bit multipliers (or half of that for 18×18 -bit multipliers), and has an **Fmax**=86.23 MHz registered performance using the TimeQuest slow 85C model.

A simulation of the function approximation is shown in Fig. 2.60. The simulation shows the result for the input value $x = 0.75/8 = 0.0938 = 3072/2^{15}$. In the shift phase the input $x = 3072$ is normalized by a **pre** factor of 8. The normalized result 24576 is in the range $x \in [0.5, 1) \approx [16\,384, 32\,768)$. Then several MAC operations are computed to arrive at $f = \sqrt{0.75} \times 2^{15} = 28\,378$. Finally a denormalization with a **post** factor of $\sqrt{2} \times 2^{13} = 11\,585$ takes place and the final result $f = \sqrt{0.75/8} \times 2^{15} = 10\,032$ is transferred to the output register.

2.31

If 8 bit plus sign precision is sufficient, we would build a square root via

$$\begin{aligned} f(x) &= \sqrt{x} = 0.3171 + 0.8801x - 0.1977x^2 \\ &= (81 + 225x - 51x^2)/256. \end{aligned} \quad (2.94)$$

Based on the fact that no coefficient is larger than 1.0 we can select a scaling factor of 256.

2.10 Fast Magnitude Approximation

In some applications such as FFT, image processing, or automatic gain control (AGC) of an incoherent receiver that uses complex data of the type $x+jy$, a fast approximation of the magnitude $r = \sqrt{x^2 + y^2}$ is needed. In AGC application the magnitude estimation is used to adjust the gain of the incoming signals in such a way that they are neither so small that large quantization noise occurs nor large that the arithmetic will overflow.

Another example is the edge detection in image processing where we like to know, based on the gradient in G_x and G_y direction, whether the overall gradient $\sqrt{G_x^2 + G_y^2}$ has crossed the threshold and is considered to be an edge. Here also not much precision is needed. In image processing we find that a very coarse estimation such as

$$r \approx |x| + |y| \quad (2.95)$$

is often used. Let us call this a zero L_0 approximation to the magnitude estimation. From the trigonometric relation $\sin(\phi) + \cos(\phi) = \sqrt{2} \sin(\phi + \pi/4)$ we see that the error of this estimation L_0 can be as large as 40%.

Much more precision is provided with the CORDIC algorithm or polynomial approximation but long delays and high resource penalties must be

expected. If the approximation should be fast, and needs somehow to be more precise than L_0 , a max/min approximation of the type

$$r \approx \alpha \max(|x|, |y|) + \beta \min(|x|, |y|) \quad (2.96)$$

can be used. The approximation will be of low latency and require only a few resources. Such an approximation will be symmetric to 45° and we may choose the factors α and β to optimize the L_1 (minimum average error) or L_∞ (minimum maximum error) norms. Table 2.14 shows the L_1 and L_∞ optimal values and popular approximations. The L_∞ approximation with $(1, 0.375)$, for instance, is used in the HSP50110 communication IC from Intersil [87]. From the effective number of bits shown in column five we see that even with full coefficient precision this is a coarse estimate and the magnitude does not have more than 5 bits precision. Figure 2.61 shows the computed magnitude approximation for these five α/β choices optimized by a linear search within a reasonable range. From a practical implementation approach the choice $\beta = 1/4$ seemed to be most interesting with low implementation effort and low average error norm L_1 .

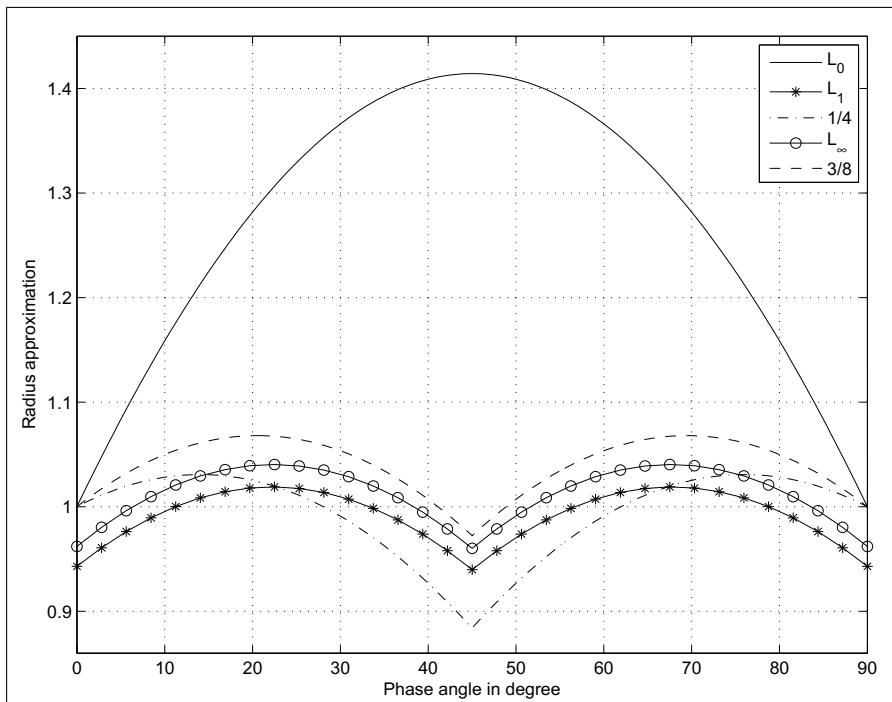


Fig. 2.61. Magnitude approximation using max/min method

Table 2.14. Coefficient characteristics of the max/min magnitude approximation. The effective bits are based on the L_∞ norm

α	β	L_1	L_∞	eff. bits	Comments
1	1	27.1%	41.4%	1.2	L_0
0.943	0.386	1.97%	6.0%	4.1	L_1 optimum
1	1/4	3.17%	11.6%	3.1	L_1 approximation
0.962	0.396	2.45%	4.0%	4.6	L_∞ optimum
1	3/8	4.22%	6.8%	3.9	L_∞ approximation

Example 2.32: VHDL Design of Magnitude Circuit

Consider the VHDL code¹³ of a 16-bit magnitude approximation.

```

PACKAGE N_bit_int IS      -- User define types
    SUBTYPE S16 IS INTEGER RANGE -2**15 TO 2**15-1;
END N_bit_int;

LIBRARY work; USE work.N_bit_int.ALL;

LIBRARY ieee;              -- Using predefined Packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
-----
ENTITY magnitude IS          -----> Interface
    PORT (clk      : IN STD_LOGIC;      -- System clock
          reset    : IN STD_LOGIC;      -- Asynchron reset
          x, y     : IN S16;           -- System inputs
          r        : OUT S16 :=0);   -- System output
END;
-----
ARCHITECTURE fpga OF magnitude IS
    SIGNAL x_r, y_r : S16 := 0;
BEGIN
    -- Approximate the magnitude via
    -- r = alpha*max(|x|,|y|) + beta*min(|x|,|y|)
    -- use alpha=1 and beta=1/4
    PROCESS(clk, reset, x, y, x_r, y_r)
    VARIABLE mi, ma, ax, ay : S16 := 0; -- temporals
    BEGIN
        IF reset = '1' THEN      -- Asynchronous clear
            x_r <= 0; y_r <= 0;
        ELSIF rising_edge(clk) THEN
            x_r <= x; y_r <= y;
        END IF;
        ax := ABS(x_r); -- Take absolute values first
        ay := ABS(y_r);
        IF ax > ay THEN -- Determine max and min values
            mi := ay;
        ELSE
            mi := ax;
        END IF;
        r := mi + (ay - mi)/4;
    END PROCESS;
END;

```

¹³ The equivalent Verilog code `mag.v` for this example can be found in Appendix A on page 812. Synthesis results are shown in Appendix B on page 881.

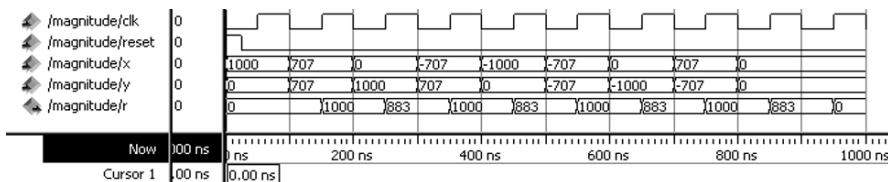


Fig. 2.62. Simulation results for the magnitude circuit.

```

        ma := ax;
ELSE
    mi := ax;
    ma := ay;
END IF;
IF reset = '1' THEN      -- Asynchronous clear
    r <= 0;
ELSIF rising_edge(clk) THEN
    r <= ma + mi/4; -- Compute r=alpha*max+beta*min
END IF;
END PROCESS;

END fpga;

```

First the input data are stored in registers. Then we take the absolute values of the input data. In the next IF condition the maximum and minimum values are determined and assigned to the variables `ma` and `mi`. Finally the $\alpha = 1$ and $\beta = 0.25$ approximation is computed and the data are stored in registers. The design runs at $F_{max}=119.59$ MHz registered performance and uses 96 LEs and no embedded multiplier. Registers are added for the input and output to measure the pipelined performance of the circuit.

The simulation result of the 16-bit pipelined magnitude computation is shown in Fig. 2.62. Eight values with angle $0, 45^\circ, 90^\circ, \dots$ are tested. Note that the computed r values are delayed due to the pipeline register and that the magnitude $r = 1000$ is approximated, but at 45° , i.e., $x = y = 1000 \cos(\pi/2) = 707$ the error becomes substantial.

2.32

Exercises

Note: If you have no prior experience with the Quartus II software, refer to the case study found in Sect. 1.4.3, p. 32. If not otherwise noted use the EP4CE115F29C7 from the Cyclone IV E family for the Quartus II synthesis evaluations.

2.1: Wallace has introduced an alternative scheme for a fast multiplier. The basic building block of this type of multiplier is a carry-save adder (CSA). A CSA takes three n -bit operands and produces two n -bit outputs. Because there is no propagation of the carry, this type of adder is sometimes called a 3:2 compress or counter.

For an $n \times n$ -bit multiplier we need a total of $n - 2$ CSAs to reduce the output to two operands. These operands then have to be added by a (fast) $2n$ -bit ripple-carry adder to compute the final result of the multiplier.

- (a) The CSA computation can be done in parallel. Determine the minimum number of levels for an $n \times n$ -bit multiplier with $n \in [0, 16]$.
- (b) Explain why, for FPGAs with fast two's complement adders, these multipliers are not more attractive than the usual array multiplier.
- (c) Explain how a pipelined adder in the final adder stage can be used to implement a faster multiplier. Use the data from Table 2.8 (p. 83) to estimate the necessary LE usage and possible speed for:
 - (c1) an 8×8 -bit multiplier
 - (c2) a 12×12 -bit multiplier

2.2: The Booth multiplier used the classical CSD code to reduce the number of necessary add/subtract operations. Starting with the LSB, typically two or three bits (called radix-4 and radix-8 algorithms) are processed in one step. The following table demonstrates possible radix-4 patterns and actions:

x_{k+1}	x_k	x_{k-1}	Accumulator activity	Comment
0	0	0	ACC \rightarrow ACC + R* (0)	within a string of "0s"
0	0	1	ACC \rightarrow ACC + R* (X)	end of a string of "1s"
0	1	0	ACC \rightarrow ACC + R* (X)	
0	1	1	ACC \rightarrow ACC + R* (2X)	end of a string of "1s"
1	0	0	ACC \rightarrow ACC + R* (-2X)	beginning of a string of "1s"
1	0	1	ACC \rightarrow ACC + R* (-X)	
1	1	0	ACC \rightarrow ACC + R* (-X)	beginning of a string of "1s"
1	1	1	ACC \rightarrow ACC + R* (0)	within a string of "1s"

The hardware requirements for a state machine implementation are an accumulator and a two's complement shifter.

- (a) Let X be a signed 6-bit two's complement representation of $-10 = 110110_2$. Complete the following table for the Booth product $P = XY = -10Y$ and indicate the accumulator activity in each step.

Step	x_5	x_4	x_3	x_2	x_1	x_0	x_{-1}	ACC	ACC + Booth rule
Start	1	1	0	1	1	0	0		
0									
1									
2									

(2.97)

- (b) Compare the latency of the Booth multiplier, with the serial/parallel multiplier for the radix-4 and radix-8 algorithms.

2.3: (a) Compile the HDL file `add2p` with the Quartus II compiler with optimization for speed and area. How many LEs are needed? Explain the results.
 (b) Conduct a simulation with $x = 32760$ and $y = 1,073,740,000$, $y = 5$, and $y = 10$.

2.4: Explain how to modify the HDL design `add1p` for subtraction.

- (a) Modify the design and simulate as an example:
 (b) $3 - 2$
 (c) $2 - 3$

2.5: (a) Develop a 8×8 serial/parallel multiplier **mul_ser** in HDL file according to (2.29) (p. 86) with the Quartus II compiler.

(b) Determine the registered performance using the **TimeQuest** slow 85C model and the used resources of the 8-bit design. What is the total multiplication latency?

2.6: Develop HDL design file **mul_ser** to multiply 12×12 -bit numbers according to (2.29) (p. 86).

(a) Simulate the new design with the values 1000×2000 .

(b) Measure the registered performance using the **TimeQuest** slow 85C model and the resources (LEs, multipliers, and M9Ks).

(c) What is the total multiplication latency of the 12×12 -bit multiplier?

2.7: (a) Design a state machine in Quartus II to implement the Booth multiplier (see Exercise 2.2) for 6×6 bit signed inputs.

(b) Simulate the four data $\pm 5 \times (\pm 9)$.

(c) Determine the registered performance using the **TimeQuest** slow 85C model.

(d) Determine LE utilization for maximum speed.

2.8: (a) Design a **generic** CSA that is used to build a Wallace-tree multiplier for an 8×8 -bit multiplier.

(b) Implement the 8×8 Wallace tree using Quartus II.

(c) Use a final adder to compute the product, and test your multiplier with a multiplication of 100×63 .

(d) Pipeline the Wallace tree. What is the maximum throughput of the pipelined design?

2.9: (a) Use the principle of component instantiation, using the predefined macros **LPM_ADD_SUB** and **LPM_MULT**, to write the VHDL code for a pipelined complex 8-bit multiplier, (i.e., $(a + jb)(c + jd) = ac - bd + j(ad + bc)$), with all operands a, b, c , and d in 8-bit.

(b) Determine the registered performance **Fmax** using the **TimeQuest** slow 85C model.

(c) Determine LE and embedded multipliers used for maximum speed synthesis.

(d) How many pipeline stages does the optimal single **LPM_MULT** multiplier have?

(e) How many pipeline stages does the optimal complex multiplier have in total if you use:

(e1) LE-based multipliers?

(e2) Embedded array multipliers?

2.10: An alternative algorithm for a complex multiplier is:

$$\begin{aligned} s[1] &= a - b & s[2] &= c - d & s[3] &= c + d \\ m[1] &= s[1]d & m[2] &= s[2]a & m[3] &= s[3]b \\ s[4] &= m[1] + m[2] & s[5] &= m[1] + m[3] \\ (a + jb)(c + jd) &= s[4] + js[5] \end{aligned} \quad (2.98)$$

which, in general, needs five adders and three multipliers. Verify that if one coefficient, say $c + jd$ is known, then $s[2], s[3]$, and d can be prestored and the algorithm reduces to three adds and three multiplications. Also

(a) Design a pipelined 5/3 complex multiplier using the above algorithm for 8-bit signed inputs. Use the predefined macros **LPM_ADD_SUB** and **LPM_MULT**.

(b) Measure the **Fmax** registered performance using the **TimeQuest** slow 85C model and the used resources (LEs, multipliers, and M9Ks) for maximum speed synthesis.

(c) How many pipeline stages does the single **LPM_MULT** multiplier have?

(d) How many pipeline stages does the complex multiplier have in total is you use

(d1) LE-based multipliers?

(d2) Embedded array multipliers?

- 2.11:** Compile the HDL file `cordic` with the Quartus II compiler, and
- Conduct a simulation (using the MODELSIM stimuli file `cordic.do`) with $x_{in} = \pm 30$ and $y_{in} = \pm 55$. Determine the radius factor for all four simulations.
 - Determine the maximum errors for radius and phase, compared with an unquantized computation.
- 2.12:** Modify the HDL design `cordic` to implement stages 4 and 5 of the CORDIC pipeline.
- Compute the rotation angle, and compile the VHDL code.
 - Conduct a simulation with values $x_{in} = \pm 30$ and $y_{in} = \pm 55$.
 - What are the maximum errors for radius and phase, compared with the unquantized computation?
- 2.13:** Consider a floating-point representation with a sign bit, $E = 7$ -bit exponent width, and $M = 10$ bits for the mantissa (not counting the hidden one).
- Compute the bias using (2.24) p. 75.
 - Determine the (absolute) largest number that can be represented.
 - Determine the (absolutely measured) smallest number (not including denormals) that can be represented.
- 2.14:** Using the result from Exercise 2.13
- Determine the representation of $f_1 = 9.25_{10}$ in this (1,7,10) floating-point format.
 - Determine the representation of $f_2 = -10.5_{10}$ in this (1,7,10) floating-point format.
 - Compute $f_1 + f_2$ using floating-point arithmetic.
 - Compute $f_1 * f_2$ using floating-point arithmetic.
 - Compute f_1/f_2 using floating-point arithmetic.
- 2.15:** For the IEEE single-precision format (see Table 2.6, p. 79) determine the 32-bit representation of:
- $f_1 = -0$.
 - $f_2 = \infty$.
 - $f_3 = 9.25_{10}$.
 - $f_4 = -10.5_{10}$.
 - $f_5 = 0.1_{10}$.
 - $f_6 = \pi = 3.141593_{10}$.
 - $f_7 = \sqrt{3}/2 = 0.8660254_{10}$.
- 2.16:** Compile the HDL file `div_res` from Example 2.18 (p. 96) to divide two numbers.
- Simulate the design with the values 234/3.
 - Simulate the design with the values 234/1.
 - Simulate the design with the values 234/0. Explain the result.
- 2.17:** Design a nonperforming divider based on the HDL file `div_res` from Example 2.18 (p. 96).
- Simulate the design with the values 234/50 as shown in Fig. 2.22, p. 99.
 - Measure the registered performance F_{max} using the TimeQuest slow 85C model, the used resources (LEs, multipliers, and M9Ks) and latency for maximum speed synthesis.
- 2.18:** Design a nonrestoring divider based on the HDL file `div_res` from Example 2.18 (p. 96).
- Simulate the design with the values 234/50 as shown in Fig. 2.23, p. 100.

(b) Measure the registered performance **F_{max}** using the **TimeQuest** slow 85C model, the used resources (LEs, multipliers, and M9Ks) and latency for maximum speed synthesis.

2.19: Shift operations are usually implemented with a barrelshifter, which can be inferred in VHDL via the **SLL** instruction. Unfortunately, the **SLL** is not supported for **STD_LOGIC** in VHDL-1993, but we can design a barrelshifter in many different ways to achieve the same function. We wish to design 12-bit barrelshifters, that have the following entity:

```
ENTITY lshift IS                                     -----> Interface
  GENERIC (W1 : INTEGER := 12; -- data bit width
           W2 : integer := 4); -- ceil(log2(W1));
  PORT (clk      : IN STD_LOGIC;
        distance : IN STD_LOGIC_VECTOR (W2-1 DOWNTO 0);
        data     : IN STD_LOGIC_VECTOR (W1-1 DOWNTO 0);
        result   : OUT STD_LOGIC_VECTOR (W1-1 DOWNTO 0));
END;
```

that should be verified via the simulation shown in Fig. 2.63. Use input and output registers for **data** and **result**, no register for the **distance**. Select one of the following devices:

- (I)** EP4CE11F29C7 from the Cyclone IV E family
- (II)** EP2C35F672C6 from the Cyclone II family
- (III)** EPM7128SLC84-7 from the MAX7000S family
- (a1)** Use a **PROCESS** and (sequentially) convert each bit of the distance vector in an equivalent power-of-two constant multiplication. Use **lshift** as the entity name.
- (a2)** Measure the registered performance **F_{max}** using the **TimeQuest** slow 85C model and the resources (LEs, multipliers, and M4Ks/M9Ks).
- (b1)** Use a **PROCESS** and shift (in a loop) the input data always 1 bit only, until the loop counter and **distance** show the same value. Then transfer the shifted data to the output register. Use **lshiftloop** as the entity name.
- (b2)** Measure the registered performance **F_{max}** using the **TimeQuest** slow 85C model and the resources (LEs, multipliers, and M4Ks/M9Ks).
- (c1)** Use a **PROCESS** environment and “demux” with a loop statement the distance vector in an equivalent multiplication factor. Then use a single (array) multiplier to perform the multiplication. Use **lshiftdemux** as the entity name.
- (c2)** Measure the registered performance **F_{max}** using the **TimeQuest** slow 85C model and the resources (LEs, multipliers, and M4Ks/M9Ks).
- (d1)** Use a **PROCESS** environment and convert with a **case** statement the distance vector to an equivalent multiplication factor. Then use a single (array) multiplier to perform the multiplication. Use **lshiftmul** as the entity name.
- (d2)** Measure the registered performance **F_{max}** using the **TimeQuest** slow 85C model and the resources (LEs, multipliers, and M4Ks/M9Ks).
- (e1)** Use the **lpm_clshift** megafunction to implement the 12-bit barrelshifter. Use **lshiftlpm** as the entity name.
- (e2)** Measure the registered performance **F_{max}** using the **TimeQuest** slow 85C model and the resources (LEs, multipliers, and M4Ks/M9Ks).
- (d)** Compare all five barrelshifter designs in terms of registered performance, resources (LEs, multipliers, and M4Ks/M9Ks), and design reuse, i.e., effort to change data width and the use of software other than Quartus II.

2.20: **(a)** Design the PREP benchmark 3 shown in Fig. 2.64a with the Quartus II software. The design is a small FSM with eight states, eight data input bits **i**, **clk**,

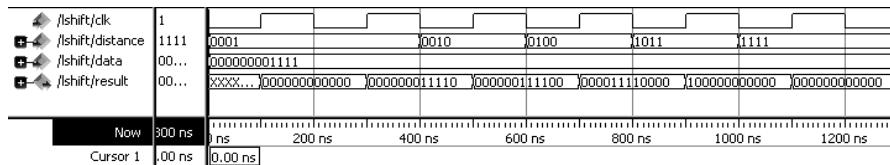


Fig. 2.63. Test bench for the barrel shifter from Exercise 2.19

rst, and an 8-bit data-out signal **o**. The next state and output logic is controlled by a positive-edge triggered **clk** and an asynchronous reset **rst**, see the simulation in Fig. 2.64c for the function test. The following table shows next state and output assignments,

Current state	Next state	i (Hex)	o (Hex)
start	start	(3c)'	00
start	sa	3c	82
sa	sc	2a	40
sa	sb	1f	20
sa	sa	(2a)'(1f)'	04
sb	se	aa	11
sb	sf	(aa)'	30
sc	sd	—	08
sd	sg	—	80
se	start	—	40
sf	sg	—	02
sg	start	—	01

where x' is the condition not x .

(b) Determine the registered performance **Fmax** using the **TimeQuest** slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for a single copy. Compile the HDL file with the synthesis **Optimization Technique** set to **Speed**, **Balanced** or **Area**; this can be found in the **Analysis & Synthesis Settings** section under **Settings** in the **Assignments** menu. Which synthesis options are optimal in terms of LE count and registered performance?

Select one of the following devices:

(b1) EP4CE115F29C7 from the Cyclone IV E family

(b2) EP2C35F672C6 from the Cyclone II family

(b3) EPM7128SLC84-7 from the MAX7000S family

(c) Design the multiple instantiation for benchmark 3 as shown in Fig. 2.64b.

(d) Determine the registered performance **Fmax** using the **TimeQuest** slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for the design with the maximum number of instantiations of PREP benchmark 3. Use the optimal synthesis option you found in (b) for the following devices:

(d1) EP4CE115F29C7 from the Cyclone IV E family

(d2) EP2C35F672C6 from the Cyclone II family

(d3) EPM7128SLC84-7 from the MAX7000S family

2.21: (a) Design the PREP benchmark 4 shown in Fig. 2.65a with the Quartus II software. The design is a large FSM with sixteen states, 40 transitions, eight data input bits **i[0..7]**, **clk**, **rst** and 8-bit data-out signal **o[0..7]**. The next state is controlled by a positive-edge-triggered **clk** and an asynchronous reset **rst**, see the

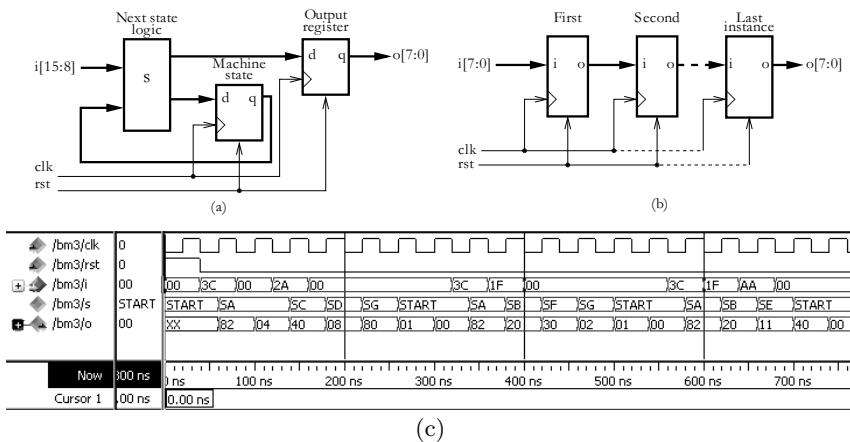


Fig. 2.64. PREP benchmark 3. (a) Single design. (b) Multiple instantiation. (c) Test bench to check function

simulation in Fig. 2.65c for a partial function test. The following shows the output decoder table:

Current state	$o[7..0]$	Current state	$o[7..0]$
s0	0 0 0 0 0 0 0 0	s1	0 0 0 0 0 1 1 0
s2	0 0 0 1 1 0 0 0	s3	0 1 1 0 0 0 0 0
s4	1 x x x x x x 0	s5	x 1 x x x x 0 x
s6	0 0 0 1 1 1 1 1	s7	0 0 1 1 1 1 1 1
s8	0 1 1 1 1 1 1 1	s9	1 1 1 1 1 1 1 1
s10	x 1 x 1 x 1 x 1	s11	1 x 1 x 1 x 1 x
s12	1 1 1 1 1 0 1	s13	1 1 1 1 0 1 1 1
s14	1 1 0 1 1 1 1 1	s15	0 1 1 1 1 1 1 1

where X is the unknown value. Note that the output values does not have an additional output register as in the PREP 3 benchmark. The next state table is:

Current state	Next state	Condition	Current state	Next state	Condition
s0	s0	$i = 0$	s0	s1	$1 \leq i \leq 3$
s0	s2	$4 \leq i \leq 31$	s0	s3	$32 \leq i \leq 63$
s0	s4	$i > 63$	s1	s0	$i0 \times i1$
s1	s3	$(i0 \times i1)'$	s2	s3	—
s3	s5	—	s4	s5	$i0 + i2 + i4$
s4	s6	$(i0 + i2 + i4)'$	s5	s5	$i0'$
s5	s7	$i0$	s6	s1	$i6 \times i7$
s6	s6	$(i6 + i7)'$	s6	s8	$i6 \times i7'$
s6	s9	$i6' \times i7$	s7	s3	$i6' \times i7'$
s7	s4	$i6 \times i7$	s7	s7	$i6 \oplus i7$
s8	s1	$(i4 \odot i5)i7$	s8	s8	$(i4 \odot i5)i7'$
s8	s11	$i4 \oplus i5$	s9	s9	$i0'$
s9	s11	$i0$	s10	s1	—
s11	s8	$i \neq 64$	s11	s15	$i = 64$
s12	s0	$i = 255$	s12	s12	$i \neq 255$
s13	s12	$i1 \oplus i3 \oplus i5$	s13	s14	$(i1 \oplus i3 \oplus i5)'$
s14	s10	$i > 63$	s14	s12	$1 \leq i \leq 63$
s14	s14	$i = 0$	s15	s0	$i7 \times i1 \times i0$
s15	s10	$i7 \times i1' \times i0$	s15	s13	$i7 \times i1 \times i0'$
s15	s14	$i7 \times i1' \times i0'$	s15	s15	$i7'$

where ik is bit k of input i , the symbol $'$ is the not operation, \times is the Boolean AND operation, $+$ is the Boolean OR operation, \odot is the Boolean equivalence operation, and \oplus is the XOR operation.

(b) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for a single copy. Compile the HDL file with the synthesis Optimization Technique set to Speed, Balanced or Area; this can be found in the Analysis & Synthesis Settings section under Settings in the Assignments menu. Which synthesis options are optimal in terms of LE count and registered performance?

Select one of the following devices:

(b1) EP4CE115F29C7 from the Cyclone IV E family

(b2) EP2C35F672C6 from the Cyclone II family

(b3) EPM7128SLC84-7 from the MAX7000S family

(c) Design the multiple instantiation for benchmark 4 as shown in Fig. 2.65b.

(d) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for the design with the maximum number of instantiations of PREP benchmark 4. Use the optimal synthesis option you found in (b) for the following devices:

(d1) EP4CE115F29C7

(d2) EP2C35F672C6

(d3) EPM7128SLC84-7

2.22: (a) Design an 8×8 -bit signed multiplier `smul8x8` using M9Ks memory blocks and the partitioning technique discussed in (2.32), p. 90.

(b) Use a short C or MATLAB script to produce the three required MIF files. You need signed/signed, signed/unsigned, and unsigned/unsigned tables. The last entry in the table should be:

(b1) 11111111 : 11100001; --> 15 * 15 = 225 for unsigned/unsigned.

(b2) 11111111 : 11110001; --> -1 * 15 = -15 for signed/unsigned.

(b3) 11111111 : 00000001; --> -1 * (-1) = 1 for signed/signed.

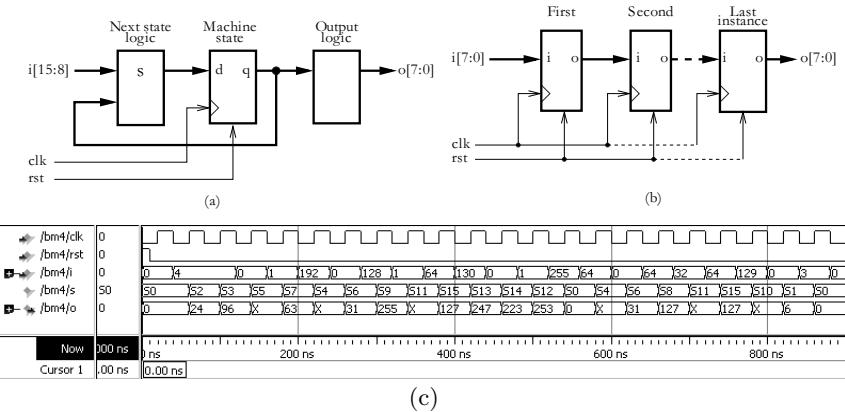


Fig. 2.65. PREP benchmark 4. (a) Single design. (b) Multiple instantiation. (c) Test bench to check function

(c) Verify the design with the three data pairs $-128 \times (-128) = 16384$; $-128 \times 127 = -16256$; $127 \times 127 = 16129$.

(d) Measure the registered performance F_{max} using the TimeQuest slow 85C model and determine the resources used.

2.23: (a) Design an 8×8 -bit additive half-square (AHSM) multiplier `ahsm8x8` as shown in Fig. 2.17, p. 91.

(b) Use a short C or MATLAB script to produce the two required MIF files. You need a 7- and 8-bit D1 encoded square tables. The first entries in the 7-bit table should be:

```
depth= 128; width = 14;
address_radix = bin; data_radix = bin;
content begin
 0000000 : 0000000000000000; --> (1_d1 * 1_d1)/2 = 0
 0000001 : 00000000000010; --> (2_d1 * 2_d1)/2 = 2
 0000010 : 00000000000100; --> (3_d1 * 3_d1)/2 = 4
 0000011 : 00000000001000; --> (4_d1 * 4_d1)/2 = 8
 0000100 : 00000000001100; --> (5_d1 * 5_d1)/2 = 12
  ...

```

(c) Verify the design with the three data pairs $-128 \times (-128) = 16384$; $-128 \times 127 = -16256$; $127 \times 127 = 16129$.

(d) Measure the registered performance F_{max} using the TimeQuest slow 85C model and determine the resources used.

2.24: (a) Design an 8×8 -bit differential half-square (DHSM) multiplier `dhsm8x8` as shown in Fig. 2.18, p. 92.

(b) Use a short C or MATLAB script to produce the two required MIF files. You need an 8-bit standard square table and a 7-bit D1 encoded table. The last entries in the tables should be:

(b1) $1111111 : 1000000000000000$; $--> (128_d1 * 128_d1)/2 = 8192$ for the 7-bit

D1 table.

(b2) $11111111 : 111111100000000 ; \rightarrow (255*255)/2 = 32512$ for the 8-bit half-square table.

(c) Verify the design with the three data pairs $-128 \times (-128) = 16384$; $-128 \times 127 = -16256$; $127 \times 127 = 16129$.

(d) Measure the registered performance F_{max} using the TimeQuest slow 85C model and determine the resources used.

2.25: (a) Design an 8×8 -bit quarter-square multiplication multiplier `qsm8x8` as shown in Fig. 2.19, p. 93.

(b) Use a short C or MATLAB script to produce the two required MIF files. You need an 8-bit standard quarter square table and an 8-bit D1 encoded quarter square table. The last entries in the tables should be:

(b1) $11111111 : 111111100000000 ; \rightarrow (255*255)/4 = 16256$ for the 8-bit quarter square table.

(b2) $11111111 : 1000000000000000 ; \rightarrow (256_d1 * 256_d1)/4 = 16384$ for the D1 8-bit quarter-square table.

(c) Verify the design with the three data pairs $-128 \times (-128) = 16384$; $-128 \times 127 = -16256$; $127 \times 127 = 16129$.

(d) Measure the registered performance F_{max} using the TimeQuest slow 85C model and determine the resources used.

2.26: Plot the function approximation and the error function as shown in Fig. 2.50a and b (p. 144) for the arctan function for $x \in [-1, 1]$ using the following coefficients:

(a) For $N = 2$ use $f(x) = 0.0000 + 0.8704x = (0 + 223x)/256$.

(b) For $N = 4$ use $f(x) = 0.0000 + 0.9857x + 0.0000x^2 - 0.2090x^3 = (0 + 252x + 0x^2 - 53x^3)/256$.

2.27: Plot the function approximation and the error function as shown, for instance, in Fig. 2.50a and b (p. 144) for the arctan function using the 8-bit precision coefficients, but with increased convergence range and determine the maximum error:

(a) For the $\arctan(x)$ approximation using coefficients from (2.64), p. 145 with $x \in [-2, 2]$

(b) For the $\sin(x)$ approximation using the coefficients from (2.79), p. 151 with $x \in [0, 2]$

(c) For the $\cos(x)$ approximation using the coefficients from (2.82), p. 152 with $x \in [0, 2]$

(d) For the \sqrt{x} approximation using the coefficients from (2.94), p. 165 with $x \in [0, 2]$

2.28: Plot the function approximation and the error function as shown, for instance, in Fig. 2.54a and b (p. 153) for the e^x function using the 8-bit precision coefficients, but with increased convergence range and determine the maximum error:

(a) For the e^x approximation using the coefficients from (2.84), p. 152 with $x \in [-1, 2]$

(b) For the e^{-x} approximation using the coefficients from (2.87), p. 154 with $x \in [-1, 2]$

(c) For the $\ln(1 + x)$ approximation using the coefficients from (2.88), p. 158 with $x \in [0, 2]$

(d) For the $\log_{10}(1 + x)$ approximation using the coefficients from (2.92), p. 159 with $x \in [0, 2]$

2.29: Plot the function approximation and the error function as shown in Fig. 2.56a and b (p. 155) for the $\ln(1+x)$ function for $x \in [0, 1]$ using the following coefficients:

(a) For $N = 2$ use $f(x) = 0.0372 + 0.6794x = (10 + 174x)/256$.

(b) For $N = 3$ use $f(x) = 0.0044 + 0.9182x - 0.2320x^2 = (1 + 235x - 59x^2)/256$.

2.30: Plot the function approximation and the error function as shown in Fig. 2.59a and b (p. 160) for the \sqrt{x} function for $x \in [0.5, 1]$ using the following coefficients:

- (a) For $N = 2$ use $f(x) = 0.4238 + 0.5815x = (108 + 149x)/256$.
- (b) For $N = 3$ use $f(x) = 0.3171 + 0.8801x - 0.1977x^2 = (81 + 225x - 51x^2)/256$

3. Finite Impulse Response (FIR) Digital Filters

3.1 Digital Filters

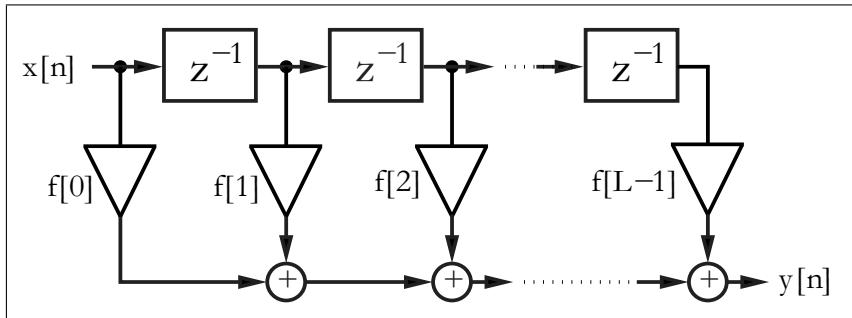
Digital filters are typically used to modify or alter the attributes of a signal in the time or frequency domain. The most common digital filter is the linear time-invariant (LTI) filter. An LTI interacts with its input signal through a process called linear convolution, denoted by $y = f * x$ where f is the filter's impulse response, x is the input signal, and y is the convolved output. The linear convolution process is formally defined by:

$$y[n] = x[n] * f[n] = \sum_k x[k]f[n - k] = \sum_k f[k]x[n - k]. \quad (3.1)$$

LTI digital filters are generally classified as being *finite impulse response* (i.e., FIR), or *infinite impulse response* (i.e., IIR). As the name implies, an FIR filter consists of a finite number of sample values, reducing the above convolution sum to a finite sum per output sample instant. An IIR filter, however, requires that an infinite sum be performed. An FIR design and implementation methodology is discussed in this chapter, while IIR filter issues are addressed in Chap. 4.

The motivation for studying digital filters is found in their growing popularity as a primary DSP operation. Digital filters are rapidly replacing classic analog filters, which were implemented using RLC components and operational amplifiers. Analog filters were mathematically modeled using ordinary differential equations of Laplace transforms. They were analyzed in the time or s (also known as Laplace) domain. Analog prototypes are now only used in IIR design, while FIR are typically designed using direct computer specifications and algorithms.

In this chapter it is assumed that a digital filter, an FIR in particular, has been designed and selected for implementation. The FIR design process will be briefly reviewed, followed by a discussion of FPGA implementation variations.

**Fig. 3.1.** Direct form FIR filter

3.2 FIR Theory

An FIR with constant coefficients is an LTI digital filter. The output of an FIR of length L or order $N = L - 1$, to an input time-series $x[n]$, is given by a *finite* version of the convolution sum given in (3.1), namely:

$$y[n] = x[n] * f[n] = \sum_{k=0}^{L-1} f[k]x[n-k], \quad (3.2)$$

where $f[0] \neq 0$ through $f[L-1] \neq 0$ are the filter's L coefficients. They also correspond to the FIR's impulse response. For LTI systems it is sometimes more convenient to express (3.2) in the z -domain with

$$Y(z) = F(z)X(z), \quad (3.3)$$

where $F(z)$ is the FIR's *transfer function* defined in the z -domain by

$$F(z) = \sum_{k=0}^{L-1} f[k]z^{-k}. \quad (3.4)$$

The length- L LTI FIR filter is graphically interpreted in Fig. 3.1. It can be seen to consist of a collection of a “tapped delay line,” adders, and multipliers. One of the operands presented to each multiplier is an FIR coefficient, often referred to as a “tap weight” for obvious reasons. Historically, the FIR filter is also known by the name “transversal filter,” suggesting its “tapped delay line” structure.

The *roots* of polynomial $F(z)$ in (3.4) define the zeros of the filter. The filter order N is the number of roots. A length- L filter has order $N = L - 1$. The presence of only zeros is the reason that FIRs are sometimes called *all zero filters*. In Chap. 5 we will discuss an important class of FIR filters (called CIC filters) that are *recursive* but also FIR. This is possible because the poles produced by the recursive part are canceled by the nonrecursive part of the filter. The effective pole/zero plot also then has *only* zeros, i.e., is an *all-zero*

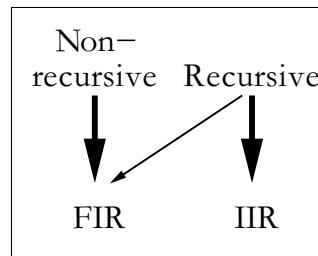


Fig. 3.2. Relation between structure and impulse length

filter or FIR. We note that nonrecursive filters are always FIR, but recursive filters can be either FIR or IIR. Figure 3.2 illustrates this dependence.

3.2.1 FIR Filter with Transposed Structure

A variation of the direct FIR model is called the *transposed FIR filter*. It can be constructed from the FIR filter in Fig. 3.1 by:

- Exchanging the input and output
- Inverting the direction of signal flow
- Substituting an adder by a fork, and vice versa

A transposed FIR filter is shown in Fig. 3.3 and is, in general, the preferred implementation of an FIR filter. The benefit of this filter is that we do not need an extra shift register for $x[n]$, and there is no need for an extra pipeline stage for the adder (tree) of the products to achieve high throughput.

The following examples show a direct implementation of the transposed filter.

Example 3.1: Programmable FIR Filter

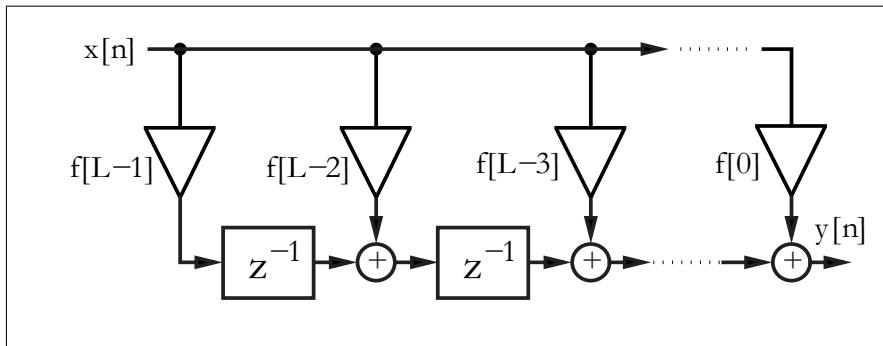


Fig. 3.3. FIR filter in the transposed structure

We recall from the discussion of sum-of-product (SOP) computations using a PDSP (see Sect. 2.8, p. 124) that, for B_x data/coefficient bit width and filter length L , additional $\log_2(L)$ bits for unsigned SOP and $\log_2(L) - 1$ guard bits for signed arithmetic must be provided. For a 9-bit signed data/coefficient and $L = 4$, the adder width must be $9 + 9 + \log_2(4) - 1 = 19$. The following VHDL code² shows the generic specification for an implementation for a length-4 filter:

```
-- This is a generic FIR filter generator
-- It uses W1 bit data/coefficients bits
LIBRARY ieee;           -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-- -----
ENTITY fir_gen IS          -----> Interface
  GENERIC (W1 : INTEGER := 9; -- Input bit width
           W2 : INTEGER := 18; -- Multiplier bit width 2*W1
           W3 : INTEGER := 19; -- Adder width = W2+log2(L)-1
           W4 : INTEGER := 11; -- Output bit width
           L  : INTEGER := 4  -- Filter length
         );
  PORT ( clk      : IN STD_LOGIC;      -- System clock
         reset    : IN STD_LOGIC;      -- Asynchron reset
         Load_x   : IN STD_LOGIC;      -- Load/run switch
         x_in     : IN STD_LOGIC_VECTOR(W1-1 DOWNTO 0);      -- System input
         c_in     : IN STD_LOGIC_VECTOR(W1-1 DOWNTO 0);      -- Coefficient data input
         y_out    : OUT STD_LOGIC_VECTOR(W4-1 DOWNTO 0));      -- System output
END fir_gen;
-- -----
ARCHITECTURE fpga OF fir_gen IS
  SUBTYPE SLVW1 IS STD_LOGIC_VECTOR(W1-1 DOWNTO 0);
  SUBTYPE SLVW2 IS STD_LOGIC_VECTOR(W2-1 DOWNTO 0);
  SUBTYPE SLVW3 IS STD_LOGIC_VECTOR(W3-1 DOWNTO 0);
  TYPE AO_L1SLVW1 IS ARRAY (0 TO L-1) OF SLVW1;
  TYPE AO_L1SLVW2 IS ARRAY (0 TO L-1) OF SLVW2;
  TYPE AO_L1SLVW3 IS ARRAY (0 TO L-1) OF SLVW3;

  SIGNAL x  : SLVW1;
  SIGNAL y  : SLVW3;
  SIGNAL c  : AO_L1SLVW1 ; -- Coefficient array
  SIGNAL p  : AO_L1SLVW2 ; -- Product array
  SIGNAL a  : AO_L1SLVW3 ; -- Adder array

BEGIN
  Load: PROCESS(clk, reset, c_in, c, x_in)
  BEGIN
    -----> Load data or coefficients
    IF reset = '1' THEN -- clear data and coefficients reg.
      
```

² The equivalent Verilog code `fir_gen.v` for this example can be found in Appendix A on page 813. Synthesis results are shown in Appendix B on page 881.

```

x <= (OTHERS => '0');
FOR K IN 0 TO L-1 LOOP
    c(K) <= (OTHERS => '0');
END LOOP;
ELSIF rising_edge(clk) THEN
IF Load_x = '0' THEN
    c(L-1) <= c_in;           -- Store coefficient in register
    FOR I IN L-2 DOWNTO 0 LOOP -- Coefficients shift one
        c(I) <= c(I+1);
    END LOOP;
ELSE
    x <= x_in;                -- Get one data sample at a time
END IF;
END IF;
END PROCESS Load;

SOP: PROCESS (clk, reset, a, p)-- Compute sum-of-products
BEGIN
    IF reset = '1' THEN -- clear tap registers
        FOR K IN 0 TO L-1 LOOP
            a(K) <= (OTHERS => '0');
        END LOOP;
    ELSIF rising_edge(clk) THEN
        FOR I IN 0 TO L-2 LOOP          -- Compute the transposed
            a(I) <= (p(I)(W2-1) & p(I)) + a(I+1); -- filter adds
        END LOOP;
        a(L-1) <= p(L-1)(W2-1) & p(L-1);      -- First TAP has
        END IF;                            -- only a register
        y <= a(0);
    END PROCESS SOP;

    -- Instantiate L multipliers
MulGen: FOR I IN 0 TO L-1 GENERATE
    p(i) <= c(i) * x;
END GENERATE;

y_out <= y(W3-1 DOWNTO W3-W4);

END fpga;

```

The first process, `Load`, is used to load the coefficient in a tapped delay line if `Load_x`=0. Otherwise, a data word is loaded into the `x` register. The second process, called `SOP`, implements the sum-of-products computation. The products `p(I)` are sign-extended by one bit and added to the previous partial SOP. Note also that all multipliers are instantiated by a `generate` statement, which allows the assignment of extra pipeline stages. Finally, the output `y_out` is assigned the value of the SOP divided by 256, because the coefficients are all assumed to be fractional (i.e., $|f[k]| \leq 1.0$). The design uses 93 LEs, four embedded multipliers, and has an `Fmax`=157.38 MHz registered performance using the `TimeQuest` slow 85C model.

To simulate this length-4 filter consider a Daubechies DB4 filter coefficient with

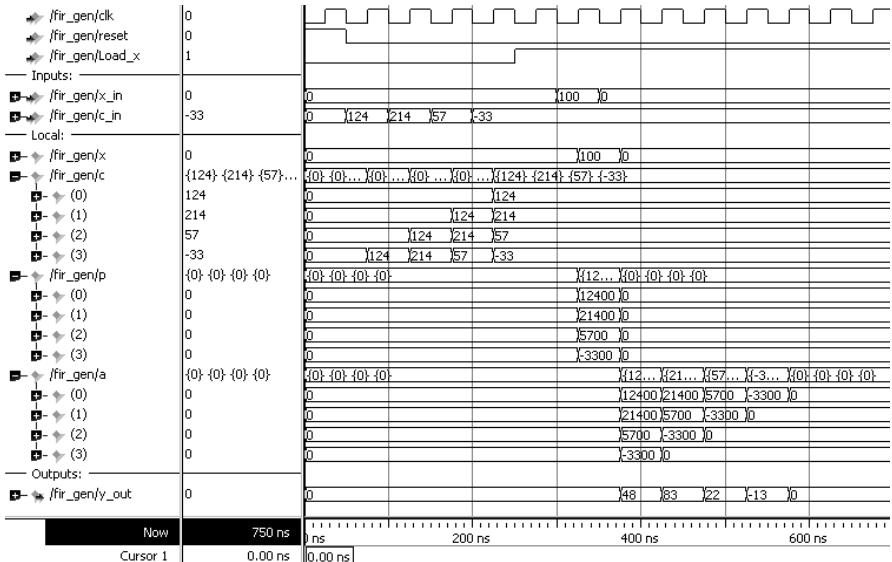


Fig. 3.4. Simulation of the 4-tap programmable FIR filter with Daubechies filter coefficients loaded

$$G(z) = \frac{(1 + \sqrt{3}) + (3 + \sqrt{3})z^{-1} + (3 - \sqrt{3})z^{-2} + (1 - \sqrt{3})z^{-3}}{4\sqrt{2}},$$

$$G(z) = 0.48301 + 0.8365z^{-1} + 0.2241z^{-2} - 0.1294z^{-3}.$$

Quantizing the coefficients to eight bits (plus a sign bit) of precision results in the following model:

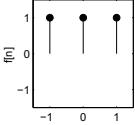
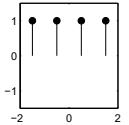
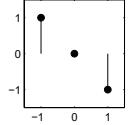
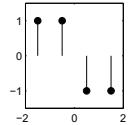
$$\begin{aligned} G(z) &= (124 + 214z^{-1} + 57z^{-2} - 33z^{-3}) / 256 \\ &= \frac{124}{256} + \frac{214}{256}z^{-1} + \frac{57}{256}z^{-2} - \frac{33}{256}z^{-3}. \end{aligned}$$

As can be seen from Fig. 3.4, in the first four steps after reset we load the coefficients $\{124, 214, 57, -33\}$ into the tapped delay line c . Then we check the impulse response of the filter by loading 100 into the x register. The embedded multiplier results $p(I)$ are ready after one clock cycle. The first valid output y_{out} is then available after 375 ns. 3.1

3.2.2 Symmetry in FIR Filters

The center of an FIR's impulse response is an important point of symmetry. It is sometimes convenient to define this point as the 0th sample instant. Such filter descriptions are *a-causal* (centered notation). For an odd-length FIR, the a-causal filter model is given by:

Table 3.1. Four possible linear-phase FIR filters $F(z) = \sum_k f[k]z^{-k}$

Symmetry L	$f[n] = f[-n]$ odd	$f[n] = f[-n]$ even	$f[n] = -f[-n]$ odd	$f[n] = -f[-n]$ even
Example				
				
Zeros at	$\pm 120^\circ$	$\pm 90^\circ, 180^\circ$	$0^\circ, 180^\circ$	$0^\circ, 2 \times 180^\circ$

$$F(z) = \sum_{k=-(L-1)/2}^{(L-1)/2} f[k]z^{-k}. \quad (3.5)$$

The FIR's frequency response can be computed by evaluating the filter's transfer function about the periphery of the unity circle, by setting $z = e^{j\omega T}$. It then follows that:

$$F(\omega) = F(e^{j\omega T}) = \sum_k f[k]e^{-jk\omega T}. \quad (3.6)$$

We then denote with $|F(\omega)|$ the filter's *magnitude frequency response* and $\phi(\omega)$ denotes the *phase response*, and satisfies:

$$\phi(\omega) = \arctan \left(\frac{\Im(F(\omega))}{\Re(F(\omega))} \right). \quad (3.7)$$

Digital filters are more often characterized by phase and magnitude than by the z -domain transfer function or the complex frequency transform.

3.2.3 Linear-phase FIR Filters

Maintaining phase integrity across a range of frequencies is a desired system attribute in many applications such as communications and image processing. As a result, designing filters that establish linear-phase versus frequency is often mandatory. The standard measure of the phase linearity of a system is the “group delay” defined by:

$$\tau(\omega) = -\frac{d\phi(\omega)}{d\omega}. \quad (3.8)$$

A perfectly linear-phase filter has a group delay that is constant over a range of frequencies. It can be shown that linear-phase is achieved if the filter is symmetric or antisymmetric, and it is therefore preferable to use the

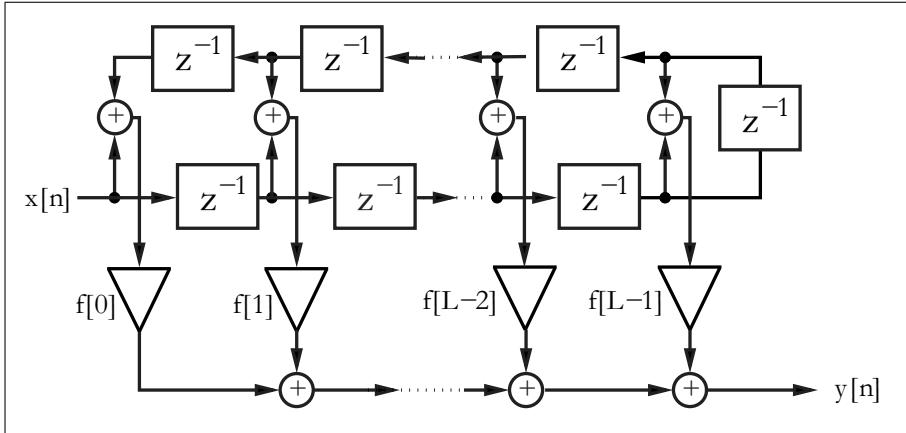


Fig. 3.5. Linear-phase filter with reduced number of multipliers

a-causal framework of (3.5). From (3.7) it can be seen that a constant group delay can only be achieved if the frequency response $F(\omega)$ is a purely real or imaginary function. This implies that the filter's impulse response possesses even or odd symmetry. That is:

$$f[n] = f[-n] \quad \text{or} \quad f[n] = -f[-n]. \quad (3.9)$$

An odd-length even-symmetry FIR filter would, for example, have a frequency response given by:

$$F(\omega) = f[0] + \sum_{k>0} f[k]e^{-jk\omega T} + f[-k]e^{jk\omega T} \quad (3.10)$$

$$= f[0] + 2 \sum_{k>0} f[k] \cos(k\omega T), \quad (3.11)$$

which is seen to be a purely real function of frequency. Table 3.1 summarizes the four possible choices of symmetry, antisymmetry, even length and odd length L . In addition, Table 3.1 graphically displays an example of each class of linear-phase FIR.

The symmetry properties intrinsic to a linear-phase FIR can also be used to reduce the necessary number of multipliers L , as shown in Fig. 3.1. Consider the linear-phase FIR shown in Fig. 3.5 (even symmetry assumed), which fully exploits coefficient symmetry. Observe that the “symmetric” architecture has a multiplier budget per filter cycle exactly half of that found in the direct architecture shown in Fig. 3.1 (L versus $L/2$) while the number of adders remains constant at $L - 1$.

3.3 Designing FIR Filters

Modern digital FIR filters are designed using computer-aided engineering (CAE) tools. The filters used in this chapter are designed using the MATLAB Signal Processing toolbox. The toolbox includes an “Interactive Lowpass Filter Design” demo example that covers many typical digital filter designs, including:

- Equiripple (also known as minimax) FIR design, which uses the Parks–McClellan and Remez exchange methods for designing a linear-phase (symmetric) equiripple FIR. This equiripple design may also be used to design a differentiator or Hilbert transformer.
- Kaiser window design using the inverse DFT method weighted by a Kaiser window.
- Least square FIR method. This filter design also has ripple in the passband and stopband, but the mean least square error is minimized.
- Four IIR filter design methods (Butterworth, Chebyshev I and II, and elliptic) which will be discussed in Chap. 4.

The FIR methods are individually developed in this section. Most often we already know the transfer function (i.e., magnitude of the frequency response) of the desired filter. Such a lowpass specification typically consists of the passband $[0 \dots \omega_p]$, the transition band $[\omega_p \dots \omega_s]$, and the stopband $[\omega_s \dots \pi]$ specification, where the sampling frequency is assumed to be 2π . To compute the filter coefficients we may therefore apply the *direct frequency* method discussed next.

3.3.1 Direct Window Design Method

The discrete Fourier transform (DFT) establishes a direct connection between the frequency and time domains. Since the frequency domain is the domain of filter definition, the DFT can be used to calculate a set of FIR filter coefficients that produce a filter that approximates the frequency response of the target filter. A filter designed in this manner is called a *direct FIR filter*. A direct FIR filter is defined by:

$$f[n] = \text{IDFT}(F[k]) = \sum_k F[k]e^{j2\pi kn/L}. \quad (3.12)$$

From basic signals and systems theory, it is known that the spectrum of a real signal is Hermitian. That is, the real spectrum has even symmetry and the imaginary spectrum has odd symmetry. If the synthesized filter should have only real coefficients, the target DFT design spectrum must therefore be Hermitian or $F[k] = F^*[-k]$, where the $*$ denotes conjugate complex.

Consider a length-16 direct FIR filter design with a rectangular window, shown in Fig. 3.6a, with the passband ripple shown in Fig. 3.6b. Note that

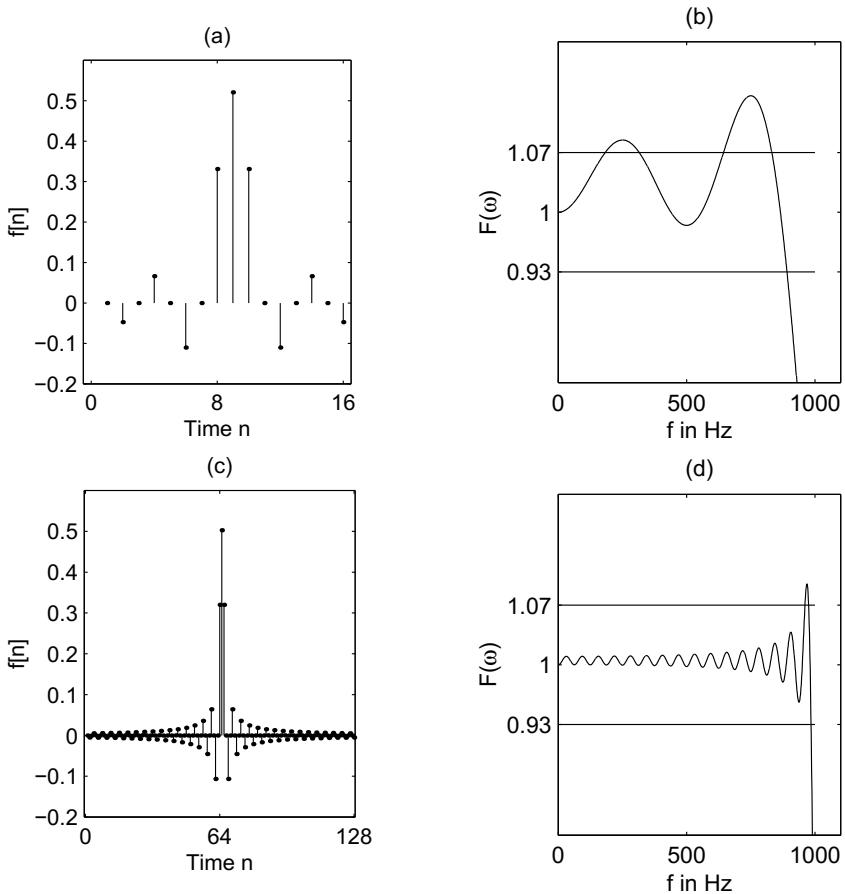


Fig. 3.6. Gibbs phenomenon. (a) Impulse response of FIR lowpass with $L = 16$. (b) Passband of transfer function $L = 16$. (c) Impulse response of FIR lowpass with $L = 128$. (d) Passband of transfer function $L = 128$

the filter provides a reasonable approximation to the ideal lowpass filter with the greatest mismatch occurring at the edges of the transition band. The observed “ringing” is due to the Gibbs phenomenon, which relates to the inability of a finite Fourier spectrum to reproduce sharp edges. The Gibbs ringing is implicit in the direct inverse DFT method and can be expected to be about $\pm 7\%$ over a wide range of filter orders. To illustrate this, consider the example filter with length 128, shown in Fig. 3.6c, with the passband ripple shown in Fig. 3.6d. Although the filter length is essentially increased (from 16 to 128) the ringing at the edge still has about the same quantity. The effects of ringing can only be suppressed with the use of a data “window” that tapers smoothly to zero on both sides. Data windows overlay the FIR’s

impulse response, resulting in a “smoother” magnitude frequency response with an attendant widening of the transition band. If, for instance, a Kaiser window is applied to the FIR, the Gibbs ringing can be reduced as shown in Fig. 3.7(upper). The deleterious effect on the transition band can also be seen. Other classic window functions are summarized in Table 3.2. They differ in terms of their ability to make tradeoffs between “ringing” and transition bandwidth extension. The number of recognized and published window functions is large. The most common windows, denoted $w[n]$, are:

- Rectangular: $w[n] = 1$
- Bartlett (triangular) : $w[n] = 2n/N$
- Hanning: $w[n] = 0.5(1 - \cos(2\pi n/L))$
- Hamming: $w[n] = 0.54 - 0.46 \cos(2\pi n/L)$
- Blackman: $w[n] = 0.42 - 0.5 \cos(2\pi n/L) + 0.08 \cos(4\pi n/L)$
- Kaiser: $w[n] = I_0(\beta \sqrt{1 - (n - L/2)^2/(L/2)^2})$

Table 3.2 shows the most important parameters of these windows.

Table 3.2. Parameters of commonly used window functions

Name	3-dB band- width	First zero	Maximum sidelobe	Sidelobe decrease per octave	Equivalent Kaiser β
Rectangular	$0.89/T$	$1/T$	-13 dB	-6 dB	0
Bartlett	$1.28/T$	$2/T$	-27 dB	-12 dB	1.33
Hanning	$1.44/T$	$2/T$	-32 dB	-18 dB	3.86
Hamming	$1.33/T$	$2/T$	-42 dB	-6 dB	4.86
Blackman	$1.79/T$	$3/T$	-74 dB	-6 dB	7.04
Kaiser	$1.44/T$	$2/T$	-38 dB	-18 dB	3

The 3-dB bandwidth shown in Table 3.2 is the bandwidth where the transfer function is decreased from DC by 3 dB or $\approx 1/\sqrt{2}$. Data windows also generate sidelobes, to various degrees, away from the 0th harmonic. Depending on the smoothness of the window, the third column in Table 3.2 shows that some windows do not have a zero at the first or second zero DFT frequency $1/T$. The maximum sidelobe gain is measured relative to the 0th harmonic value. The fifth column describes the asymptotic decrease of the window per octave. Finally, the last column describes the value β for a Kaiser window that emulates the corresponding window properties. The Kaiser window, based on the first order Bessel function I_0 , is special in two respects. It is nearly optimal in terms of the relationship between “ringing” suppression and transition width, and second, it can be tuned by β , which determines the ringing of the filter. This can be seen from the following equation credited to Kaiser:

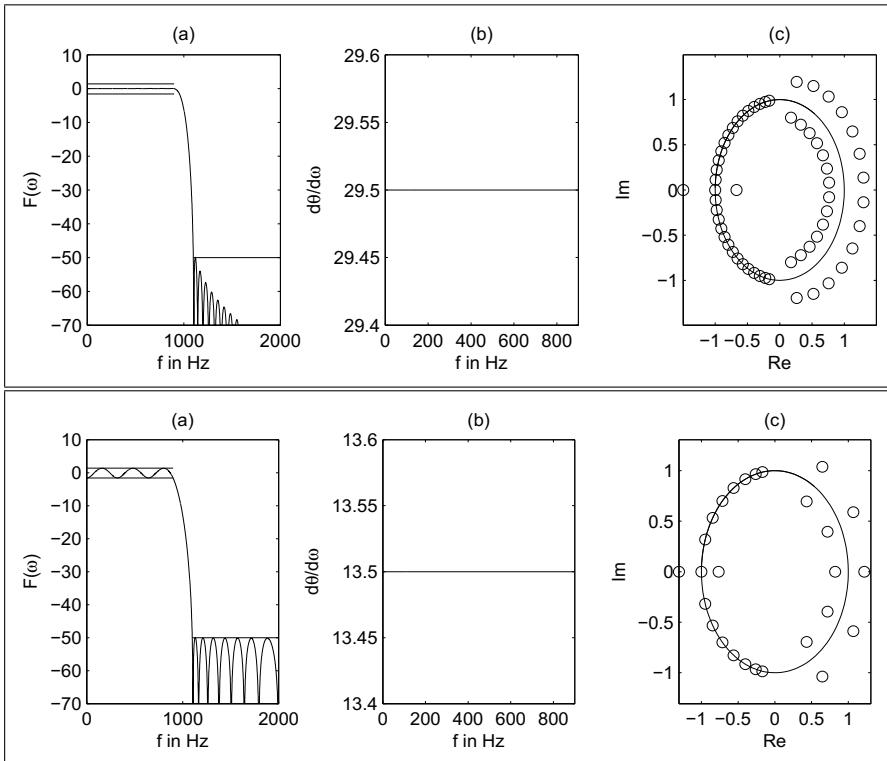


Fig. 3.7. (upper) Kaiser window design with order 59. (lower) Parks-McClellan design with order 27.

(a) Transfer function. (b) Group delay of passband. (c) Zero plot

$$\beta = \begin{cases} 0.1102(A - 8.7) & A > 50, \\ 0.5842(A - 21)^{0.4} + 0.07886(A - 21) & 21 \leq A \leq 50, \\ 0 & A < 21, \end{cases} \quad (3.13)$$

where $A = 20 \log_{10} \varepsilon_r$ is both stopband attenuation and the passband ripple in dB. The Kaiser window length to achieve a desired level of suppression can be estimated:

$$L = \frac{A - 8}{2.285(\omega_s - \omega_p)} + 1. \quad (3.14)$$

The length is generally correct within an error of ± 2 taps.

3.3.2 Equiripple Design Method

A typical filter specification not only includes the specification of passband ω_p and stopband ω_s frequencies and ideal gains, but also the allowed deviation (or ripple) from the desired transfer function. The transition band is

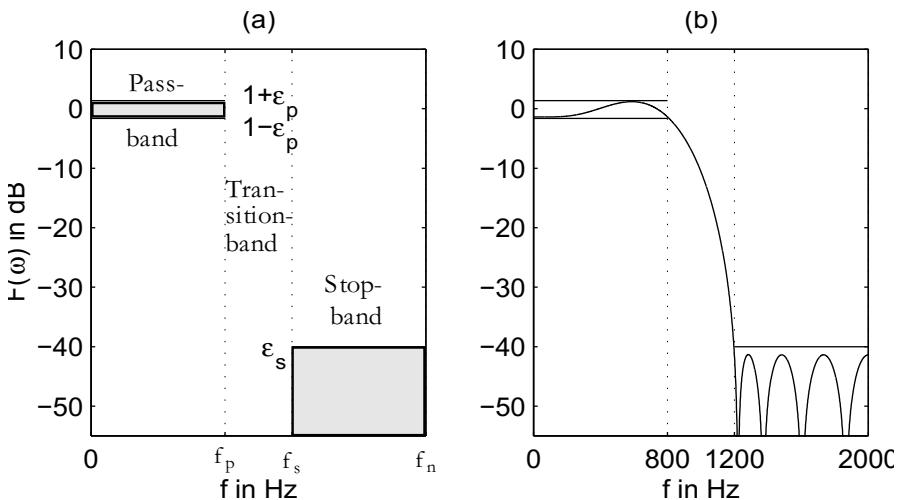


Fig. 3.8. Parameters for the filter design. (a) Tolerance scheme (b) Example function, which fulfills the scheme. (f_p passband frequency; f_s stopband frequency; f_n Nyquist frequency, i.e., 1/2 times the sampling frequency)

most often assumed to be arbitrary in terms of ripples. A special class of FIR filter that is particularly effective in meeting such specifications is called the equiripple FIR. An equiripple design protocol minimizes the maximal deviations (ripple error) from the ideal transfer function. The equiripple algorithm applies to a number of FIR design instances. The most popular are:

- Lowpass filter design (in MATLAB³ use `firpm(N,F,A,W)`), with tolerance scheme as shown in Fig. 3.8a
- Hilbert filter, i.e., a unit magnitude filter that produces a 90° phase shift for all frequencies in the passband (in MATLAB use `firpm(N, F, A, 'Hilbert')`)
- Differentiator filter that has a linear increasing frequency magnitude proportional to ω (in MATLAB use `firpm(N,F,A,'differentiator')`)

The equiripple or minimum-maximum algorithm is normally implemented using the *Parks–McClellan iterative method*. The Parks–McClellan method is used to produce a equiripple or minimax data fit in the frequency domain. It is based on the “alternation theorem” that says that there is exactly one polynomial, a Chebyshev polynomial with minimum length, that fits into a given tolerance scheme. Such a tolerance scheme is shown in Fig. 3.8a, and Fig. 3.8b shows a polynomial that fulfills this tolerance scheme. The length of the polynomial, and therefore the filter, can be estimated for a lowpass with

³ In previous MATLAB versions the function `remez` had to be used.

$$L = \frac{-10 \log_{10}(\varepsilon_p \varepsilon_s) - 13}{2.324(\omega_s - \omega_p)} + 1, \quad (3.15)$$

where ε_p is the passband and ε_s the stopband ripple.

The algorithm iteratively finds the location of locally maximum errors that deviate from a nominal value, reducing the size of the maximal error per iteration, until all deviation errors have the same value. Most often, the *Remez method* is used to select the new frequencies by selecting the frequency set with the largest peaks of the error curve between two iterations, see [88, p. 478]. This is why the MATLAB equiripple function was called `remez` in the past (now renamed to `firpm` for Parks-McClellan).

Compared to the *direct frequency method*, with or without data windows, the advantage of the equiripple design method is that passband and stopband deviations can be specified differently. This may, for instance, be useful in audio applications where the ripple in the passband may be specified to be higher, because the ear only perceives differences larger than 3 dB.

We note from Fig. 3.7(lower) that the equiripple design having the same tolerance requirements as the Kaiser window design enjoys a considerably reduced filter order, i.e., 27 compared with 59.

3.4 Constant Coefficient FIR Design

There are only a few applications (e.g., adaptive filters) where we need a general programmable filter architecture like the one shown in Example 3.1 (p. 182). In many applications, the filters are LTI (i.e., linear time invariant) and the coefficients do not change over time. In this case, the hardware effort can essentially be reduced by exploiting the multiplier and adder (trees) needed to implement the FIR filter arithmetic.

With available digital filter design software the production of FIR coefficients is a straightforward process. The challenge remains to map the FIR design into a suitable architecture. The direct or transposed forms are preferred for maximum speed and lowest resource utilization. Lattice filters are used in adaptive filters because the filter can be enlarged by one section, without the need for recomputation of the previous lattice sections. But this feature only applies to PDSPs and is less applicable to FPGAs. We will therefore focus our attention on the direct and transposed implementations. We will start with possible improvements to the direct form and will then move on to the transposed form. At the end of the section we will discuss an alternative design approach using distributed arithmetic.

3.4.1 Direct FIR Design

The direct FIR filter shown in Fig. 3.1 (p. 180) can be implemented in VHDL using (sequential) `PROCESS` statements or by “component instantiations” of

the adders and multipliers. A **PROCESS** design provides more freedom to the synthesizer, while component instantiation gives full control to the designer. To illustrate this, a length-4 FIR will be presented as a **PROCESS** design. Although a length-4 FIR is far too short for most practical applications, it is easily extended to higher orders and has the advantage of a short compiling time. The linear-phase (therefore symmetric) FIR's impulse response is assumed to be given by

$$f[k] = \{-1.0, 3.75, 3.75, -1.0\}. \quad (3.16)$$

These coefficients can be directly encoded into a 5-bit fractional number. For example, 3.75_{10} would have a 5-bit binary representation 011.11_2 where “.” denotes the location of the binary point. Note that it is, in general, more efficient to implement only *positive* CSD coefficients, because positive CSD coefficients have fewer nonzero terms and we can take the sign of the coefficient into account when the summation of the products is computed. See also the first step in the RAG algorithm 3.4 discussed later, p. 198.

In a practical situation, the FIR coefficients are obtained from a computer design tool and presented to the designer as floating-point numbers. The performance of a fixed-point FIR, based on floating-point coefficients, needs to be verified using simulation or algebraic analysis to ensure that design specifications remain satisfied. In the above example, the floating-point numbers are 3.75 and 1.0, which can be represented exactly with fixed-point numbers, and the check can be skipped.

Another issue that must be addressed when working with fixed-point designs is protecting the system from *dynamic range overflow*. Fortunately, the *worst-case* dynamic range growth G of an length- L FIR is easy to compute and it is:

$$G \leq \log_2 \left(\sum_{k=0}^{L-1} |f[k]| \right). \quad (3.17)$$

The total bit width is then the sum of the input bit width and the bit growth G . For the above filter for (3.16) we have $G = \log_2(9.5) < 4$, which states that the system's internal data registers need to have at least four more integer bits than the input data to insure no overflow. If 8-bit internal arithmetic is used the input data should be bounded by $\pm 128/9.5 = \pm 13$.

Example 3.2: Four-tap Direct FIR Filter

The VHDL design⁴ for a filter with coefficients $\{-1, 3.75, 3.75, -1\}$ is shown in the following listing:

```
PACKAGE n_bit_int IS      -- User defined types
    SUBTYPE S8 IS INTEGER RANGE -128 TO 127;
    TYPE A0_3S8 IS ARRAY (0 TO 3) OF S8;
END n_bit_int;
```

⁴ The equivalent Verilog code `fir.srg.v` for this example can be found in Appendix A on page 814. Synthesis results are shown in Appendix B on page 881.

```

LIBRARY work;
USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
-----
ENTITY fir_srg IS                                     -----> Interface
  PORT (clk    : IN STD_LOGIC; -- System clock
        reset   : IN STD_LOGIC; -- Asynchronous reset
        x       : IN S8;        -- System input
        y       : OUT S8);     -- System output
END fir_srg;
-----
ARCHITECTURE fpga OF fir_srg IS

  SIGNAL tap : A0_3S8;    -- Tapped delay line of bytes

BEGIN

  P1: PROCESS(clk, reset, x, tap)      -----> Behavioral Style
  BEGIN
    IF reset = '1' THEN    -- clear shift register
      FOR K IN 0 TO 3 LOOP
        tap(K) <= 0;
      END LOOP;
      y <= 0;
    ELSIF rising_edge(clk) THEN
      -- Compute output y with the filter coefficients weight.
      -- The coefficients are [-1 3.75 3.75 -1].
      -- Division for Altera VHDL is only allowed for
      -- powers-of-two values!
      y <= 2 * tap(1) + tap(1) + tap(1) / 2 + tap(1) / 4
        + 2 * tap(2) + tap(2) + tap(2) / 2 + tap(2) / 4
        - tap(3) - tap(0);
      FOR I IN 3 DOWNTO 1 LOOP
        tap(I) <= tap(I-1); -- Tapped delay line: shift one
      END LOOP;
    END IF;
    tap(0) <= x;           -- Input in register 0
  END PROCESS;
END fpga;

```

The design is a literal interpretation of the direct FIR architecture found in Fig. 3.1 (p. 180). The design is applicable to both symmetric and asymmetric filters. The output of each tap of the tapped delay line is multiplied by the appropriately weighted binary value and the results are added. The impulse response y of the filter to an impulse 10 is shown in Fig. 3.9.

3.2

There are three obvious actions that can improve this design:

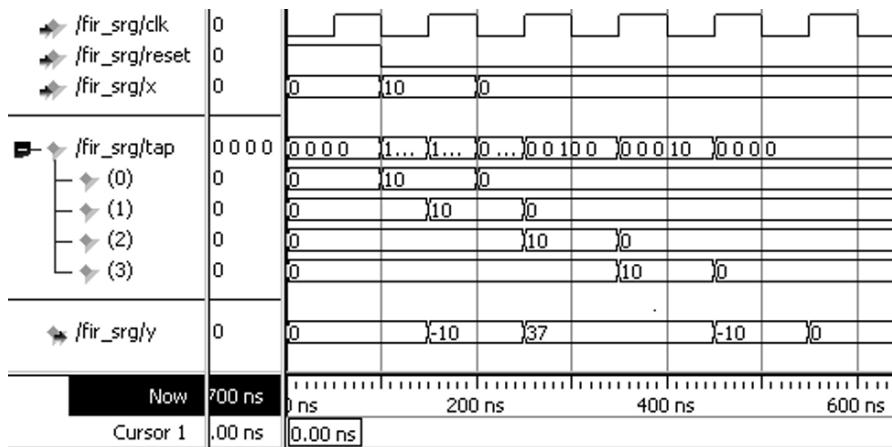


Fig. 3.9. VHDL simulation results of the FIR filter with impulse input 10

Table 3.3. Improved FIR filter

Symmetry	no	yes	no	no	yes	yes
CSD	no	no	yes	no	yes	yes
Tree	no	no	no	yes	no	yes
Speed/MHz	86.31	154.66	106.62	234.85	149.86	243.01
Size/LEs	117	98	64	146	57	82

- 1) Realize each filter coefficient with an optimized CSD code (see Chap. 2, Example 2.1, p. 62).
- 2) Increase effective multiplier speed by pipelining. The output adder should be arranged in a pipelined balance tree. If the coefficients are coded as “powers-of-two,” the pipelined multiplier and the adder tree can be merged. Pipelining has low overhead due to the fact that the LE registers are otherwise often unused. A few additional pipeline registers may be necessary if the number of terms in the tree to be added is not a power of two.
- 3) For symmetric coefficients, the multiplication complexity can be reduced as shown in Fig. 3.5 (p. 186).

The first two actions are applicable to all FIR filters, while the third applies only to linear-phase (symmetric) filters. These ideas will be illustrated by example designs.

Example 3.3: Improved Four-tap Direct FIR Filter

The design from the previous example can be improved using a CSD code for the coefficients $3.75 = 2^2 - 2^{-2}$. In addition, symmetry and pipelining can also be employed to enhance the filter’s performance. Table 3.3 shows the maximum throughput that can be expected for each different design. CSD

coding and symmetry result in smaller, more compact designs. Improvements in registered performance are obtained by pipelining the multiplier and providing an adder tree for the output accumulation. Two additional pipeline registers (i.e., 16 LEs) are necessary, however. The most compact design is expected using symmetry and CSD coding without the use of an adder tree. The partial VHDL code for producing the filter output y is shown below.

```
t1 <= tap(1) + tap(2); -- Using symmetry
t2 <= tap(0) + tap(3);
IF rising_edge(clk) THEN
    y <= 4 * t1 - t1 / 4 - t2; Apply CSD code and add
    ...

```

The fastest design is obtained when all three enhancements are used. The partial VHDL code, in this case, becomes:

```
WAIT UNTIL clk = '1'; -- Pipelined all operations
t1 <= tap(1) + tap(2); -- Use symmetry of coefficients
t2 <= tap(0) + tap(3); -- and pipeline adder
t3 <= 4 * t1 - t1 / 4; -- Pipelined CSD multiplier
t4 <= -t2;           -- Build a binary tree and add delay
y <= t3 + t4;
...

```

3.3

Exercise 3.7 (p. 221) discusses the implementation of the filter in more detail.

Direct Form Pipelined FIR Filter

Sometimes a single coefficient has more pipeline delay than all the other coefficients. We can model this delay by $f[n]z^{-d}$. If we now add a positive delay with

$$f[n] = z^d f[n]z^{-d} \quad (3.18)$$

the two delays are eliminated. Translating this into hardware means that for the direct form FIR filter we have to use the output of the d position previous register.

This principle is shown in Fig. 3.10a. Figure 3.10b shows an example of rephasing a pipelined multiplier that has two delays.

3.4.2 FIR Filter with Transposed Structure

A variation of the direct FIR filter is called the *transposed filter* and has been discussed in Sect. 3.2.1 (p. 181). The transposed filter enjoys, in the case of a constant coefficient filter, the following two additional improvements compared with the direct FIR:

- Multiple use of the repeated coefficients using the reduced adder graph (RAG) algorithm [36–39]

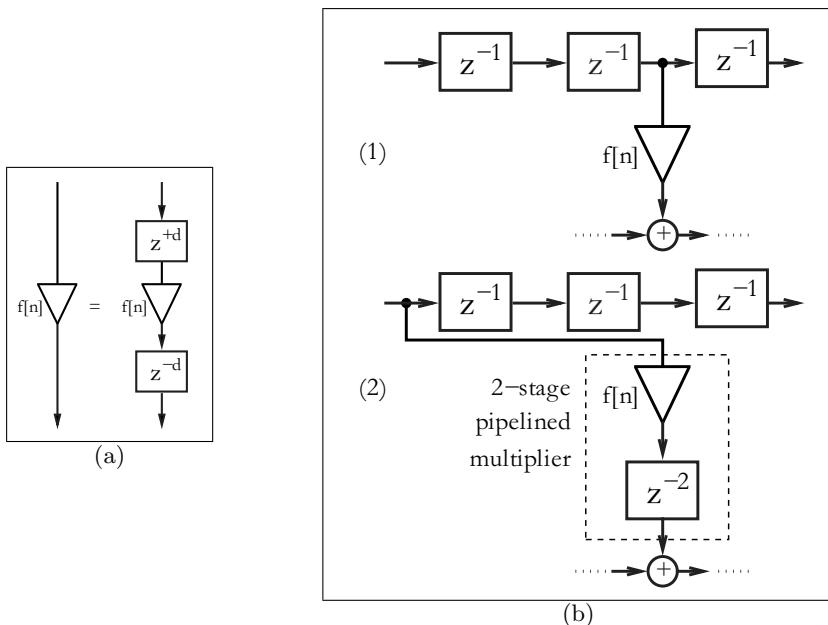


Fig. 3.10. Rephasing FIR filter. (a) Principle. (b) Rephasing a multiplier. (1) Without pipelining. (2) With two-stage pipelining

- Pipeline adders using a carry-save adder

The pipeline adder increases the speed, at additional adder and register costs, while the RAG principle will reduce the size (i.e., number of LEs) of the filter and sometimes also increase the speed. The pipeline adder principle has been discussed in Chap. 2 and here we will focus on the RAG algorithm.

In Chap. 2 it was noted that it can sometimes be advantageous to implement the factors of a constant coefficient, rather than implement the CSD code directly. For example, the CSD code realization of the constant multiplier coefficient 93 requires three adders, while the factors 3×31 only requires two adders; see Fig. 2.3 (p. 65). For a transposed FIR filter, the probability is high that all the coefficients will have several factors in common. For instance, the coefficients 9 and 11 can be built using $8 + 1 = 9$ for the first and $11 = 9 + 2$ for the second. This reduces the total effort by one adder. In general, however, finding the optimal reduced adder graph (RAG) is an *NP-hard* problem. As a result, heuristics must be used. The RAG algorithm first suggested by Dempster and Macleod is described next [38].

Algorithm 3.4: **Reduced Adder Graph**

- 1) Reduce all coefficients in the input set to positive odd fundamentals (OF).
- 2) Evaluate the single-coefficient adder cost of each coefficient using the MAG Table 2.3, p. 68.
- 3) Remove from the input set all power-of-two values and repeated fundamentals.
- 4) Create a graph set of all coefficients that can be built with one adder. Remove these coefficients from the input set.
- 5) Check if a pair of fundamentals in the graph set can be used to generate a coefficient in the input set by using a single adder.
- 6) Repeat step 5 until no further coefficients are added to the graph set. This completes the optimal part of the algorithm. Next follows the heuristic part of the algorithm:
- 7) Add the smallest coefficient requiring two adders (if found) from the input set and its smallest NOF. The OF and one NOF (i.e., auxiliary coefficient) requires two adders using the fundamentals in the graph set.
- 8) Go to step 5 since the two new fundamentals from step 7 can be used to build other coefficients from the input set.
- 9) Add the smallest adder cost-3 or higher OF to the graph set and use the minimum NOF sum for this coefficient.
- 10) Go to step 5 until all coefficients are synthesized.

Steps 1–6 are straightforward, but steps 7–10 are potentially complex since the number of theoretical graphs increases exponentially. To simplify the process it is helpful to use the MAG coding data shown in Table 2.3 (p. 68). Let us briefly review some of the RAG steps that are not so obvious at first glance.

In step 1 all coefficients are reduced to positive odd fundamentals (i.e., power-of-two factors are removed from each coefficient), since this maximizes the number of partial sums, and the negative signs of the coefficients are implemented in the output adder TAPs of the filter. The two coefficient -7 and $28 = 4 \times 7$ would be merged. This works fine except for the unlikely case when all coefficients are negative. Then a sign complement operation has to be added to the filter output.

In step 5 all sums of two extended fundamentals are considered. It may happen that a final division is also required, i.e., $g = (2^u f_1 \pm 2^v f_2)/2^w$. Note that multiplication or division by two can be implemented by left and right shift, respectively, i.e., they do not require hardware resources. For instance the coefficient set $\{7, 105, 53\}$ MAG coding required one, two, and three adders, respectively. In RAG the set is synthesized as $7 = 8 - 1$; $105 = 7 \times 15$; $53 = (105 + 1)/2$, requiring only three adders but also a divide/right shift operation.

In step 7 an adder cost-2 coefficient is added and the algorithm selects the auxiliary coefficient, called the non-output fundamental (NOF), with the smallest values. This is motivated by the fact that an additional small NOF will generate more additional coefficients than a larger NOF. For instance, let us assume that the coefficient 45 needs to be added and we must decide which NOF value has to be used. The NOF LUTs lists possible NOFs as 3, 5, 9, or 15. It can now be argued that, if 3 is selected, more coefficients are generated than if any other NOF is used, since 3, 6, 12, 24, 48, ... can be generated without additional effort from NOF 3. If 15 is used, for instance, as the NOF the coefficients 15, 30, 45, ..., are generated, which produces significantly fewer coefficients than NOF 3.

To illustrate the RAG algorithm, consider coding the coefficients defining the F6 half-band FIR filter of Goodman and Carey [89].

Example 3.5: Reduced Adder Graph for an F6 Half-Band Filter

The half-band filter F6 has four nonzero coefficients, namely $f[0]$, $f[1]$, $f[3]$, and $f[5]$, which are 346, 208, -44, and 9. For a first cost estimation we convert the decimal values (index 10) into binary representations (index 2) and look up the cost for the coefficients in Table 2.3 (p. 68):

$f[k]$	Cost
$f[0] = 346_{10} = 2 \times 173 = 101011010_2$	4
$f[1] = 208_{10} = 2^4 \times 13 = 11010000_2$	2
$f[3] = -44_{10} = -2^2 \times 11 = -101100_2$	2
$f[5] = 9_{10} = 3^2 = 1001_2$	1
Total	9

For the direct CSD code realization, nine adders are required. The RAG algorithms proceeds as follows:

Step	To be realized	Already realized	Action
0)	{346, 208, -44, 9}	{ - }	Initialization
1a)	{346, 208, 44, 9}	{ - }	No negative coefficients
1b)	{173, 13, 11, 9}	{ - }	Remove 2^k factors
2)	{173, 13, 11, 9}	{ - }	Look-up coefficients costs: {3, 2, 2, 1}
3)	{173, 13, 11, 9}	{ - }	Remove cost-0 coefficients from set
4)	{173, 13, 11}	{ 9 }	Realize cost-1 coefficients: 9 = 8 + 1
5)	{173, 13, 11}	{ 9, 11, 13 }	Build 11 = 9 + 2 and 13 = 9 + 4

Apply the heuristic to the remaining coefficients, starting with the coefficient with the lowest cost and smallest value. It follows that:

Step	Realize	Already realized	Action
7)	{ - }	{ 9, 11, 13, 173 }	Find representation Add NOF 3: 173 = 11 × 16 - 3

Figure 3.11 shows the resulting reduced adder graph. The number of adders is reduced from 9 to 5. The adder path delay is also reduced from 4 to 3. 3.5

A program `ragopt.exe` that implements the optimal part of the algorithms can be found in the book CD under `util`. Compared with the orig-

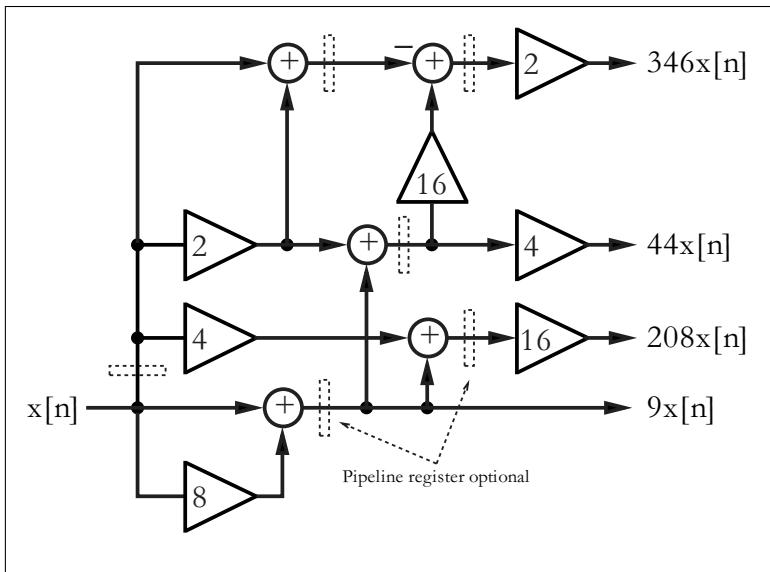


Fig. 3.11. Realization of F6 using RAG algorithm

inal algorithm only some minor improvements have been reported over the years [90].

- The MAG LUT table used has been extended to 14 bits (Gustafsson et al. [91] have actually extended the cost table to 19 bits but do not keep the fundamental table) and all 32 MAG adder cost-4 graph are now considered when computing the minimum NOF sum. Within 14 bits only two coefficients (i.e., 14 709, 15 573) are of cost 5 and, as long as these coefficients are not used, the computed minimum NOF sum list will be optimal in the RAG-95 sense.
- In step 7 all adder cost-2 graph are now considered. There are three such graphs, i.e., a single fundamental followed by an adder cost-2 factor, a sum of two fundamentals, and an adder cost-1 factor or a sum of three fundamentals.
- The last improvement is based on the adder cost-2 selection, which sometimes produced suboptimal results in the RAG-95 algorithm when multiple adder cost-2 coefficients have to be implemented. This can be explained as follows. While the selection of the smallest NOF is motivated by the statistical observation this may lead to suboptimal results. For instance, for the coefficient set {13, 59, 479} the minimum NOFs values used by RAG-95 are {3, 5, 7} because $13 = 4 \times 3 + 1$; $59 = 64 - 5$; $479 = 59 \times 8 + 7$, resulting in a six-adder graph. If the NOF {15} is chosen instead, then all coefficients ($13 = 15 - 2$; $59 = 15 \times 4 - 1$; $479 = 15 \times 32 - 1$) benefit and RAG-05 only requires four adders, a 30% improvement. Therefore, instead of selecting

the smallest NOF for the smallest adder cost-2 coefficient, a search for the best NOF is done over all adder cost-2 coefficients.

These modifications have been implemented in the RAG-05 algorithm, while the original RAG will be called RAG-95 based on the year of publishing the algorithms.

Although the RAG algorithm has been in use for quite some time, a large set of reliable benchmark data that can be verified and reproduced by anyone was not produced until recently [90]. In a recent paper by Wang and Roy [92], for instance, 60% of the comparison RAG data were declared “unknown.” A benchmark should cover filters used in practical applications that are widely published or can easily be computed – a generation of random number filter coefficients that: (a) cannot be verified by a third party, and (b) are of no practical relevance (although used in many publications) are less useful. The problem with a RAG benchmark is that the heuristic part may give different results depending on the exact software implementation or the NOF table used. In addition, since some filters are rather long, a benchmark that lists the whole RAG is not practical in most cases. It is therefore suggested to use a benchmark based on the following equivalence transformation (remembering that the number of output fundamentals is equivalent to the number of adders required):

Theorem 3.6: RAG Equivalent Transformation

Let S_1 be a coefficient set that can be synthesized by the RAG algorithm with a set of F_1 output fundamentals and N_1 non-output fundamentals, (i.e., internal auxiliary coefficients). A congruent RAG is synthesized if a coefficient set S_2 is used that contains as fundamentals both output and non-output fundamentals from the first set $S_2 = F_1 \cup N_1$.

Proof: Assume that S_2 is synthesized via the RAG algorithm. Now all fundamentals can be synthesized with exactly one adder, since all fundamentals are synthesized in the optimal part of the algorithm. As a result a minimum number $C_2 = \#F_1 + \#N_1$ of adders for this fundamental set is used. If now set S_1 is synthesized and generates the same fundamentals (output and non-output) as set S_2 , the resulting RAG also uses the minimum number of adders. Since both use the minimum number of adders they must be congruent. q.e.d.

A corollary of Theorem 3.6 is that graphs can now be classified as (guaranteed) optimal and heuristic graphs. An optimal graph has no more than one NOF, while a heuristic graph has more than one NOF. It is only required to provide a list of the NOFs to describe a unique OF graph. If this NOF is added to the coefficient set, all OFs are synthesized via the optimal part of the algorithm, which can easily be programmed. The program `ragopt.exe` that implements the optimal part of the algorithms is in fact available on the book CD under `util`. Some example benchmarks are given in Table 3.4. The first column shows the filter name, followed by the filter length L , and the bit width of the largest coefficient B . Then the reference adder data for

Table 3.4. Required number of adders for the CSD, CSE, and RAG algorithms for lowpass filters. Prototype filters are from Goodman and Carey [89], Samueli [93], and Lim and Parker [94]

Filter name	L	B	CSD adder	CSE adder	#OF	#NOF	RAG-05 adder	NOF values
F5	11	8	6	-	3	0	3	-
F6	11	9	9	-	4	1	5	3
F7	11	9	7	-	3	1	4	23
F8	15	10	10	-	5	2	7	11, 17
F9	19	13	14	-	5	2	7	13, 1261
S1	25	9	11	6	6	0	6	-
S2	60	14	57	29	26	0	26	-
L1	121	17	145	57	51	1	52	49
L2	63	13	49	23	22	0	22	-
L3	36	11	16	5	5	0	5	-

CSD coding and common sub-expression (CSE) coding follows. The idea of the CSE coding is studied in Exercises 3.4 and 3.5 (p. 220). Note that the number of CSD adders given already takes advantage of coefficient symmetry, i.e., $f(k) = f(L - k)$. CSE required adder data are used from [92]. For the RAG algorithm the output fundamental (OF) and non-output fundamental (NOF) for RAG-2005 are listed. Note that the number of OFs is already much smaller than the filter length L . We then list in column 8 the adders required in the improved RAG-2005 version. Finally in the last column we list the NOF values that are required to synthesize the RAG filter via the optimal part of the RAG algorithms that is the basis for the program `ragopt.exe`⁵ on the book CD under `util`. `ragopt.exe` uses a MAG LUT `mag14.dat` to determine the MAG costs, and produces two output files: `firXX.dat` that contains the filter data, and a file `ragopt.pro` that has the RAG- n coefficient equations. A `grep` command for lines that start with `Build` yields the equations necessary to construct the RAG- n graph.

It can be seen that the examples from Samueli [93] and Lim and Parker [94] all produce optimal RAG results, i.e., have a maximum of one NOF. Notice particularly for long filters the improvement of RAG compared to CSD and CSE adders. Filters F5-F9 are from the Goodman and Carey set of half-band filters (see Table 5.3, p. 339) and give better results using RAG-05 than RAG-95. The benchmark data from Samueli, and Lim and Parker work very well for RAG since the filters are lowpass and therefore taper smoothly to zero at both sides, improving the likelihood of cost-1 output fundamentals.

⁵ You need to copy the program to your hard drive first; you cannot start it from the CD directly.

A more-challenging RAG design for DFT coefficients will be discussed in Chap. 6.

Pipelined RAG FIR Filter

Due to the logic delay in the RAG running through several adders, the resulting register performance of the design is not very high even for a small graph. To improve the register performance one can take advantage of the register embedded in each LE that would not otherwise be used. A single register placed at the output of an adder does therefore not require any additional logic resource. However, power-of-two coefficients that are implemented by shifting the register input word require an additional register not included in the zero-pipeline design. This design with one pipeline stage already enjoys a speed improvement of 50% compared with the non-pipelined design; see Table 3.5(Pipeline stages=1). For the fully pipelined design we need to have the same delay for each incoming path of the adders. For the F6 design one needs to build:

$$\begin{aligned}x9 &\leq 8 \times x + x; \quad \text{has delay 1} \\x11 &\leq x9 + 2 \times x \times z^{-1}; \quad \text{has delay 2} \\x13 &\leq x9 + 4 \times x \times z^{-1}; \quad \text{has delay 2} \\x3 &\leq xz^{-1} + 2 \times x \times z^{-1}; \quad \text{has delay 2} \\x173 &\leq 16 \times x11 - x3; \quad \text{has delay 3}\end{aligned}$$

i.e., one extra pipeline register is used for input x , and a maximum delay of three pipeline stage is needed. The pipelined graph is shown in Fig. 3.11 with the dashed register active. Now the coefficients in the RAG are all fully pipelined. Now we need to take care of the different delays of the coefficients. We basically have two options: we can add to the output of *all* coefficients an additional delay, that we achieve the same delay for all coefficients (three in the case of the F6 filter) and then do not need to change the output tap delay line structure; alternative we can use pipeline retiming, i.e., the multiplier outputs need to be aligned in the tap delay line according to their pipeline stages. This is a similar approach to that used in the direct FIR (see Fig. 3.10, p. 197) by aligning the coefficient adder location according to the delay, and is shown in Fig. 3.12. Note in order to build only two input adder, we had to use an additional register to delay the $x13$ coefficient.

For this half-band filter design the pipeline retiming synthesis results shown in Table 3.5 reveal that the design now runs about twice as fast with a moderate (13%) increase in LEs when compared with the unpipelined design. Since the overall cost measured by LEs/Fmax is improved, fully pipelined designs should be preferred.

Table 3.5. F6 pipeline options for the RAG algorithm

Pipeline stages	LEs	Fmax (MHz)	Cost LEs/Fmax
0	224	152.7	1.47
1	237	199.76	1.19
max	254	319.49	0.79
Gain 0/max	-13%	109%	86%

3.4.3 FIR Filters Using Distributed Arithmetic

A completely different FIR architecture is based on the distributed arithmetic (DA) concept introduced in Sect. 2.8.1 (p. 125). In contrast to a conventional sum-of-products architecture, in distributed arithmetic we always compute the sum of products of a specific bit b over *all* coefficients in one step. This is computed using a small table and an accumulator with a shifter. A signed DA filter will require a signed accumulator. To illustrate, consider the three-coefficient FIR with coefficients $\{1, -3, 7\}$ found in Example 2.25 from Chap. 2 (p. 129).

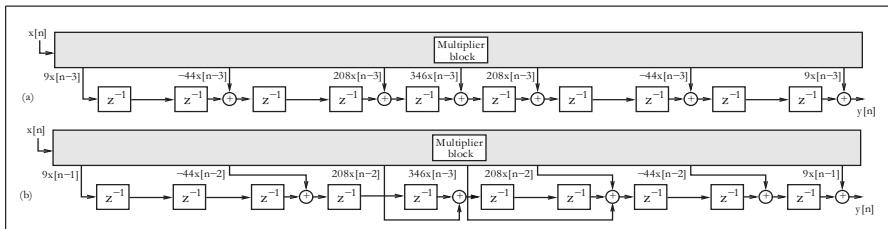
Example 3.7: Signed DA FIR Filter

For the signed DA filter, an additional state is required. See the variable `count` to process the sign bit. The following VHDL code⁶ shows the signed DA filter implementation:

```
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
```

```
PACKAGE n_bits_int IS          -- User defined types
  SUBTYPE U3 IS INTEGER RANGE 0 TO 7;
  SUBTYPE S4 IS INTEGER RANGE -8 TO 7;
  SUBTYPE S7 IS INTEGER RANGE -64 TO 63;
  SUBTYPE SLV4 IS STD_LOGIC_VECTOR(3 DOWNTO 0);
END n_bits_int;
```

⁶ The equivalent Verilog code `dasign.v` for this example can be found in Appendix A on page 817. Synthesis results are shown in Appendix B on page 881.

**Fig. 3.12.** F6 RAG filter with pipeline retiming

```

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;          -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY dasign IS           -----> Interface
    PORT (clk      : IN STD_LOGIC;   -- System clock
          reset     : IN STD_LOGIC;   -- Asynchronous reset
          x0_in    : IN SLV4;        -- First system input
          x1_in    : IN SLV4;        -- Second system input
          x2_in    : IN SLV4;        -- Third system input
          lut      : OUT S4;        -- DA look-up table
          y         : OUT S7);      -- System output

END dasign;

ARCHITECTURE fpga OF dasign IS

COMPONENT case3s      -- User defined components
    PORT ( table_in : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
           table_out : OUT INTEGER RANGE -2 TO 4);
END COMPONENT;

TYPE STATE_TYPE IS (ini, run);
SIGNAL state       : STATE_TYPE;
SIGNAL table_in   : STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL x0, x1, x2 : SLV4;
SIGNAL table_out  : INTEGER RANGE -2 TO 4;

BEGIN

    table_in(0) <= x0(0); -- Connect register to look-up table
    table_in(1) <= x1(0);
    table_in(2) <= x2(0);

P1:PROCESS (reset, clk)      -----> DA in behavioral style
    VARIABLE p : S7;           -- Temporary product register
    VARIABLE count : U3;       -- Counter for shifts
BEGIN
    IF reset = '1' THEN           -- asynchronous reset
        state <= ini;
        x0 <= (OTHERS => '0');
        x1 <= (OTHERS => '0');
        x2 <= (OTHERS => '0');
        p := 0; y <= 0;
    ELSIF rising_edge(clk) THEN
        CASE state IS
            WHEN ini =>           -- Initialization step
                state <= run;
                count := 0;
                p := 0;
                x0 <= x0_in;

```

```

x1 <= x1_in;
x2 <= x2_in;
WHEN run =>          -- Processing step
  IF count = 4 THEN -- Is sum of product done?
    y <= p;          -- Output of result to y and
    state <= ini;   -- start next sum of product
  ELSE
    IF count = 3 THEN      -- Subtract for last
      p := p / 2 - table_out * 8; -- accumulator step
    ELSE
      p := p / 2 + table_out * 8; -- Accumulation for
    END IF;             -- all other steps
    FOR k IN 0 TO 2 LOOP    -- Shift bits
      x0(k) <= x0(k+1);
      x1(k) <= x1(k+1);
      x2(k) <= x2(k+1);
    END LOOP;
    count := count + 1;
    state <= run;
  END IF;
END CASE;
END IF;
END PROCESS;

LC_Table0: case3s
  PORT MAP(table_in => table_in, table_out => table_out);

  lut <= table_out; -- Extra test signal

END fpga;

```

As suggested in Chap. 2, a shift/accumulator is used, which shifts only one position to the right for each step, instead of shifting k positions to the left. The LE table (component `case3s.vhd`) was generated using the program `dagen.exe`. The VHDL code⁷ is shown below:

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY case3s IS
  PORT ( table_in : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
         table_out : OUT INTEGER RANGE -2 TO 4);
END case3s;

ARCHITECTURE LEs OF case3s IS
BEGIN

  -- This is the DA CASE table for
  -- the 3 coefficients: -2, 3, 1
  -- automatically generated with dagen.exe -- DO NOT EDIT!

  PROCESS (table_in)

```

⁷ The equivalent Verilog code `case3s.v` for this example can be found in Appendix A on page 819. Synthesis results are shown in Appendix B on page 881.

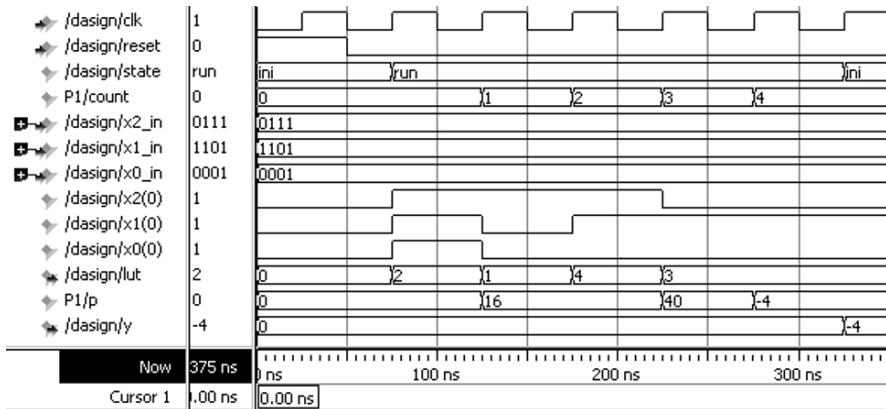


Fig. 3.13. Simulation of the 3-tap signed FIR filter with input $\{1, -3, 7\}$

```

BEGIN
  CASE table_in IS
    WHEN "000" => table_out <= 0;
    WHEN "001" => table_out <= -2;
    WHEN "010" => table_out <= 3;
    WHEN "011" => table_out <= 1;
    WHEN "100" => table_out <= 1;
    WHEN "101" => table_out <= -1;
    WHEN "110" => table_out <= 4;
    WHEN "111" => table_out <= 2;
    WHEN OTHERS => table_out <= 0;
  END CASE;
END PROCESS;
END LEs;

```

Figure 3.13 shows the simulation for the input sequence $\{1, -3, 7\}$. The simulation shows the `clk`, `reset`, `state`, and `count` signals followed by the four input signals. Next the three bits selected from the input word to address the prestored DA LUT are shown. The LUT output values $\{2, 1, 4, 3\}$ are then weighted and accumulated to generate the final output value $y = 2 + 1 \times 2 + 4 \times 4 - 3 \times 8 = -4$. The design uses 52 LEs, no embedded multiplier, and has an `Fmax`=258.4 MHz registered performance using the TimeQuest slow 85C model.

3.7

By defining the distributed arithmetic table with a `CASE` statement, the synthesizer will use logic cells to implement the LUT. This will result in a fast and efficient design only if the tables are small. For large tables, alternative means must be found. In this case, we may use the 9-kbit embedded memory blocks (M9Ks), which (as discussed in Chap. 1) can be configured as $2^{10} \times 9$, $2^{11} \times 4$, $2^{12} \times 2$ or $2^{13} \times 1$ tables. These design paths are discussed in more detail in the following.

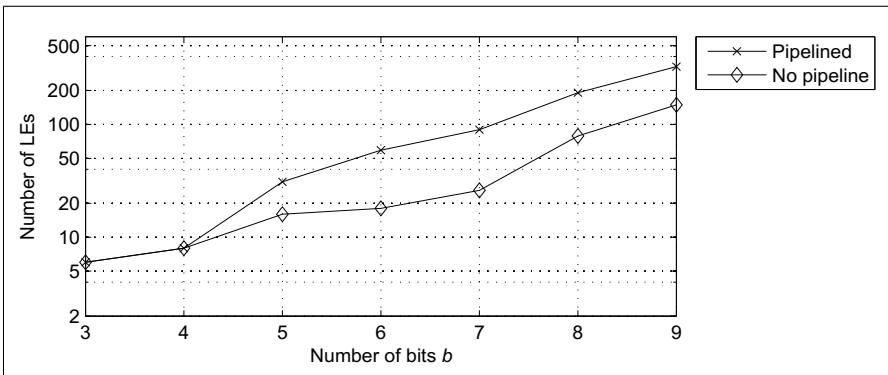


Fig. 3.14. Size comparison of synthesis results for different coding using the **CASE** statement with b input and outputs

Distributed Arithmetic Using Logic Cells

The DA implementation of an FIR filter is particularly attractive for low-order cases due to LUT address space limitations (e.g., $L \leq 4$). It should be remembered, however, that FIR filters are linear filters. This implies that the outputs of a collection of low-order filters can be added together to define the output of a high-order FIR, as shown in Fig. 2.40 (p. 132). Based on the LEs found in a Cyclone IV device, namely $2^4 \times 1$ -bit tables, a DA table for four coefficients can be implemented. The number of necessary LEs increases exponentially with order. Typically, the number of LEs is much higher than the number of M9Ks. For example, an EP2C115 contains 115K LEs but only 432 M9Ks. Also, M9Ks can be used to efficiently implement RAMs and FIFOs and other high-valued functions. It is therefore sometimes desirable to use M9Ks economically. On the other side if the design is implemented using larger tables with a $2^b \times b$ CASE statement, inefficient designs can result. The pipelined $2^9 \times 9$ table implemented with one VHDL CASE statement only, for example, required over 100 LEs. Figure 3.14 shows the number of LEs necessary for tables having three to nine bits inputs and outputs using the CASE statement generated with utility program `dagen.exe`.

Another alternative is the design using 4-input LUT only via a CASE statements, and implementing table with more than 4 inputs with an additional (binary tree) multiplexer using $2 \rightarrow 1$ multiplexer only. In this model it is straightforward to add additional pipeline registers to the modular design. For maximum speed, a register must be introduced behind each LUT and $2 \rightarrow 1$ multiplexer. This will, most likely, yield a higher LE count⁸ compared to the minimization of the one large LUT. The following example illustrates the structure of a 5-input table.

⁸ A 16:1 multiplexer is reported with 11 LEs while we need $8 + 4 + 2 + 1 = 15$ LEs for a 2:1 MUX in a tree structure [34].

Example 3.8: Five-Input DA Table

The utility program `dagen.exe` accepts filter length and coefficients, and returns the necessary PROCESS statements for the 4-input CASE table followed by a multiplexer. The VHDL output for an arbitrary set of coefficients, namely {1, 3, 5, 7, 9}, is given⁹ in the following listing:

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY case5p IS
    PORT ( clk         : IN STD_LOGIC;
           table_in   : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
           table_out  : OUT INTEGER RANGE 0 TO 25);
END case5p;

ARCHITECTURE LEs OF case5p IS

    SIGNAL lsbs : STD_LOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL msbs0 : STD_LOGIC_VECTOR(1 DOWNTO 0);
    SIGNAL table0out00, table0out01 : INTEGER RANGE 0 TO 25;

BEGIN

    -- These are the distributed arithmetic CASE tables for
    -- the 5 coefficients: 1, 3, 5, 7, 9
    -- automatically generated with dagen.exe -- DO NOT EDIT!

    PROCESS
    BEGIN
        WAIT UNTIL clk = '1';
        lsbs(0) <= table_in(0);
        lsbs(1) <= table_in(1);
        lsbs(2) <= table_in(2);
        lsbs(3) <= table_in(3);
        msbs0(0) <= table_in(4);
        msbs0(1) <= msbs0(0);
    END PROCESS;

    PROCESS      -- This is the final DA MPX stage.
    BEGIN        -- Automatically generated with dagen.exe
        WAIT UNTIL clk = '1';
        CASE msbs0(1) IS
            WHEN '0' =>    table_out <=  table0out00;
            WHEN '1' =>    table_out <=  table0out01;
            WHEN OTHERS => table_out <=  0;
        END CASE;
    END PROCESS;

    PROCESS      -- This is the DA CASE table 00 out of 1.
    BEGIN        -- Automatically generated with dagen.exe

```

⁹ The equivalent Verilog code `case5p.v` for this example can be found in Appendix A on page 815. Synthesis results are shown in Appendix B on page 881.

```

WAIT UNTIL clk = '1';
CASE lsbs IS
    WHEN "0000" => table0out00 <= 0;
    WHEN "0001" => table0out00 <= 1;
    WHEN "0010" => table0out00 <= 3;
    WHEN "0011" => table0out00 <= 4;
    WHEN "0100" => table0out00 <= 5;
    WHEN "0101" => table0out00 <= 6;
    WHEN "0110" => table0out00 <= 8;
    WHEN "0111" => table0out00 <= 9;
    WHEN "1000" => table0out00 <= 7;
    WHEN "1001" => table0out00 <= 8;
    WHEN "1010" => table0out00 <= 10;
    WHEN "1011" => table0out00 <= 11;
    WHEN "1100" => table0out00 <= 12;
    WHEN "1101" => table0out00 <= 13;
    WHEN "1110" => table0out00 <= 15;
    WHEN "1111" => table0out00 <= 16;
    WHEN OTHERS => table0out00 <= 0;
END CASE;
END PROCESS;

PROCESS      -- This is the DA CASE table 01 out of 1.
BEGIN        -- Automatically generated with dagen.exe
    WAIT UNTIL clk = '1';
    CASE lsbs IS
        WHEN "0000" => table0out01 <= 9;
        WHEN "0001" => table0out01 <= 10;
        WHEN "0010" => table0out01 <= 12;
        WHEN "0011" => table0out01 <= 13;
        WHEN "0100" => table0out01 <= 14;
        WHEN "0101" => table0out01 <= 15;
        WHEN "0110" => table0out01 <= 17;
        WHEN "0111" => table0out01 <= 18;
        WHEN "1000" => table0out01 <= 16;
        WHEN "1001" => table0out01 <= 17;
        WHEN "1010" => table0out01 <= 19;
        WHEN "1011" => table0out01 <= 20;
        WHEN "1100" => table0out01 <= 21;
        WHEN "1101" => table0out01 <= 22;
        WHEN "1110" => table0out01 <= 24;
        WHEN "1111" => table0out01 <= 25;
        WHEN OTHERS => table0out01 <= 0;
    END CASE;
END PROCESS;
END LEs;

```

The five inputs produce two **CASE** tables and a $2 \rightarrow 1$ bus multiplexer. The multiplexer may also be realized with a component instantiation using the LPM function **busmux**. The program **dagen.exe** writes a VHDL file with the name **caseX.vhd**, where X is the filter length that is also the input bit width. The file **caseXp.vhd** is the same table, except with additional pipeline registers. The component can be used directly in a state machine design or in an unrolled filter structure.

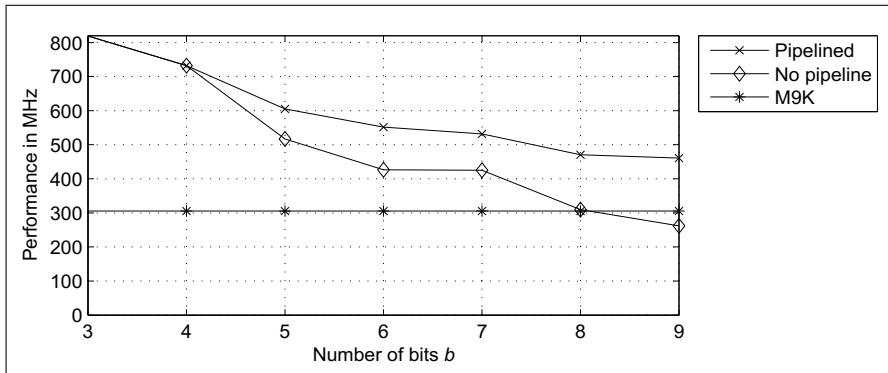


Fig. 3.15. Speed comparison for different coding styles using the CASE statement

Referring to Fig. 3.14, it can be seen that the structured VHDL code improves on the number of required LEs. Figure 3.15 compares the different design methods in terms of speed. We notice that the busmux generated VHDL code allows to run all pipelined designs with high speed outperforming the M9Ks by nearly a factor two. Without pipeline stages the synthesis tools is capable to reduce the LE count essentially, but registered performance is also reduced. Note that still a busmux design is used. The synthesis tool is not able to optimize one (large) case statement in the same way. Although we get a high registered performance using eight pipeline stages for a $2^9 \times 9$ table the design may now be too large for some applications. We may also consider the partitioning technique (Exercise 3.6, p. 221), shown in Fig. 2.39 (p. 131), or implementation with an M9K, discussed next.

DA Using Embedded Array Blocks

As mentioned in the last section, it is not economical to use the 9-kbit M9Ks memory blocks for a short FIR filter, mainly because the number of available M9Ks is limited. Also, the maximum registered speed of an M9K is 305 MHz, and an LE table implementation may be faster.

But M9Ks have only a single address decoder and if we implement a $2^3 \times 3$ table, a complete M9K would be consumed unnecessarily, and it cannot be used elsewhere. For longer filters, however, the use of M9Ks is attractive because:

- M9Ks have registered throughput at a constant 305 MHz, and
- Routing effort is reduced

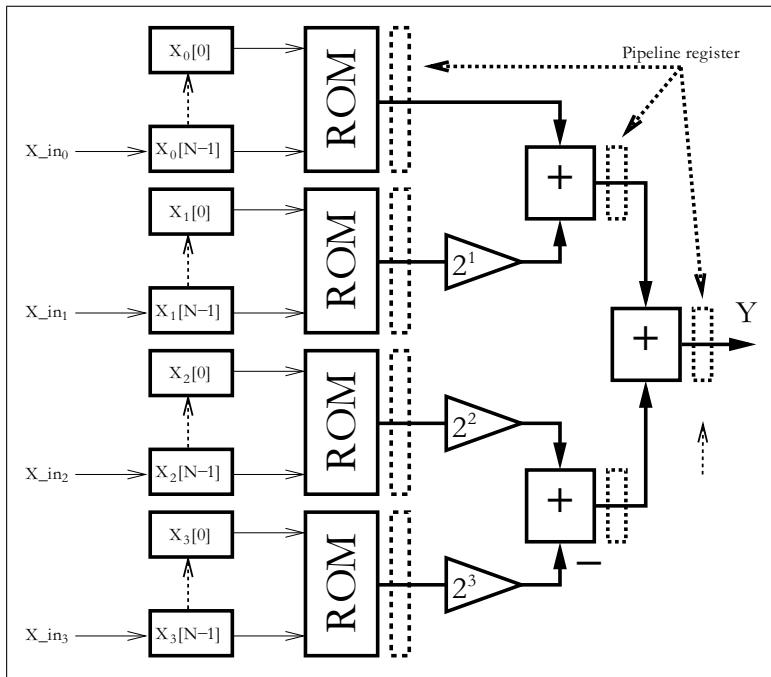


Fig. 3.16. Parallel implementation of a distributed arithmetic FIR filter

Accelerated DA Filter

To accelerate a DA filter, unrolled loops can be used. The input is applied sample by sample (one word at a time), in a bit-parallel form. In this case, for each bit of input a separate table is required. While the table size varies (input bit width equals number of filter taps), the contents of the tables are the same. The obvious advantage is a reduction of VHDL code size, if we use a component definition for the LE tables, as previously presented. To demonstrate, the unrolling of the 3-coefficients, 4-bit input example, previously considered, is developed below.

Example 3.9: Loop Unrolling for DA FIR Filter

In a typical FIR application, the input values are processed in word parallel form (i.e., see Fig. 3.16). The following VHDL code³ illustrates the unrolled DA code, according to Fig. 3.16:

```

LIBRARY ieee;          -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
-----
ENTITY dapara IS
    -----> Interface

```

³ The equivalent Verilog code `dapara.v` for this example can be found in Appendix A on page 819. Synthesis results are shown in Appendix B on page 881.

```

PORT (clk : IN STD_LOGIC;          -- System clock
      reset : IN STD_LOGIC;         -- Asynchronous reset
      x_in : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
                                -- System input
      y : OUT INTEGER RANGE -46 TO 44 := 0); -- System output
END dapara;
-----
ARCHITECTURE fpga OF dapara IS

TYPE SLVO_3B3 IS ARRAY (0 TO 3) OF
                           STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL x : SLVO_3B3;
SUBTYPE S4 IS INTEGER RANGE -8 TO 7;
TYPE AO_3S4 IS ARRAY (0 TO 3) OF S4;
SIGNAL h : AO_3S4;
SIGNAL s0 : S4;
SIGNAL s1 : S4;
SIGNAL t0, t1, t2, t3 : S4;
COMPONENT case3s
  PORT ( table_in : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
         table_out : OUT INTEGER RANGE -2 TO 4);
END COMPONENT;

BEGIN

PROCESS(clk, reset, x_in, h) ----> DA in behavioral style
BEGIN
  IF reset = '1' THEN           -- asynchronous clear
    FOR k IN 0 TO 3 LOOP
      x(k) <= (OTHERS => '0');
    END LOOP;
    y <= 0;
  t0 <= 0; t1 <= 0; t2 <= 0; t3 <= 0; s0 <= 0; s1 <= 0;
  ELSIF rising_edge(clk) THEN
    FOR l IN 0 TO 3 LOOP -- For all four vectors
      FOR k IN 0 TO 1 LOOP -- shift all bits
        x(l)(k) <= x(l)(k+1);
      END LOOP;
    END LOOP;
    FOR k IN 0 TO 3 LOOP -- Load x_in in the
      x(k)(2) <= x_in(k); -- MSBs of the registers
    END LOOP;
    y <= h(0) + 2 * h(1) + 4 * h(2) - 8 * h(3);
  -- Pipeline register and adder tree
  --   t0 <= h(0); t1 <= h(1); t2 <= h(2); t3 <= h(3);
  --   s0 <= t0 + 2 * t1; s1 <= t2 - 2 * t3;
  --   y <= s0 + 4 * s1;
  END IF;
END PROCESS;

LC_Tables: FOR k IN 0 TO 3 GENERATE -- One table for each
LC_Table: case3s                  -- bit in x_in
           PORT MAP(table_in => x(k), table_out => h(k));

```

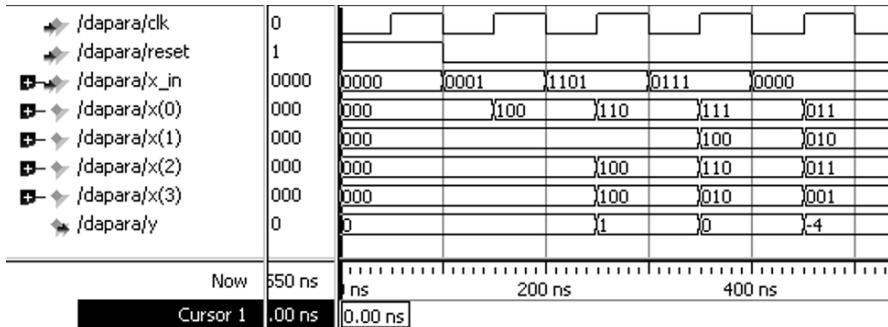


Fig. 3.17. Simulation results for the parallel distributed arithmetic FIR filter

```
END GENERATE;
```

```
END fpga;
```

The design uses four tables of size $2^3 \times 4$ and all tables have the *same* content as the table in Example 3.7 (p. 204). Figure 3.17 shows the simulation for the input sequence $\{1, -3, 7\}$. Because the input is applied serially (and bit-parallel) the expected result $-4_{10} = 1111100_2$ is computed at the 400-ns interval.

3.9

The previous design requires no embedded multiplier, 39 LEs, no M9K memory block, and runs at 205.17 MHz. An important advantage of the DA concept, compared with the general-purpose MAC design, is that pipelining is easily achieved. We can add additional pipeline registers to the table output and at the adder-tree output with no cost. To compute y , we replace the line

```
y <= h(0) + 2 * h(1) + 4 * h(2) - 8 * h(3);
```

In a first step we only pipeline the adders. We use the signals $s0$ and $s1$ for the pipelined adder within the PROCESS statement, i.e.,

```
s0 <= h(0) + 2 * h(1); s1 <= h(2) - 2 * h(3);
y <= s0 + 4 * s1;
```

and the registered performance increases to 368.60 MHz, and about the same number of LEs are used. For a fully pipeline version we also need to store the case LUT output in registers; the partial VHDL code then becomes:

```
t0 <= h(0); t1 <= h(1); t2 <= h(2); t3 <= h(3);
s0 <= t0 + 2 * t1; s1 <= t2 - 2 * t3;
y <= s0 + 4 * s1;
```

The size of the design increases to 47 LEs, because the registers of the LE that hold the case tables can no longer be used for the x input shift register. But the registered performance increases from 205.17 MHz to 420 MHz.

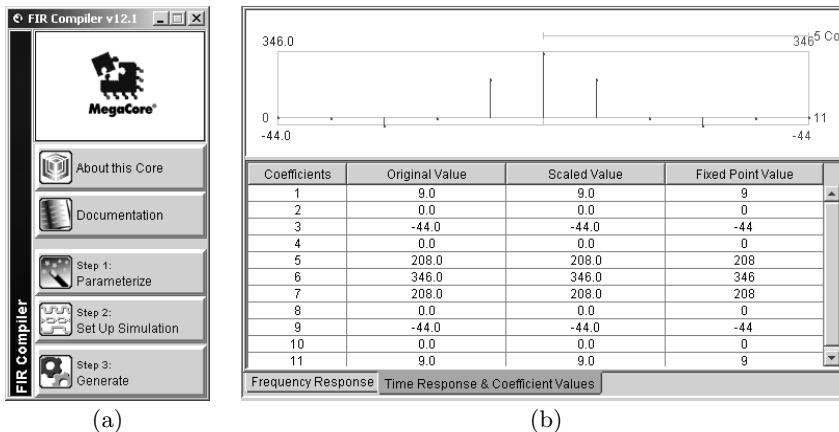


Fig. 3.18. IP design of FIR (a) IP toolbench. (b) Coefficient specification

3.4.4 IP Core FIR Filter Design

Altera and Xilinx usually also offer with the full subscription an FIR filter generator, since this is one of the most often used intellectual property (IP) blocks. For an introduction to IP blocks see Sect. 1.4.4, p. 40.

FPGA vendors in general prefer distributed arithmetic (DA)-based FIR filter generators since these designs are characterized by:

- fully pipelined architecture
- short compile time
- good resource estimation
- area results independent from the coefficient values, in contrast to the RAG algorithm

DA-based filters do not require any coefficient optimization or the computation of a RAG graph, which may be time consuming when the coefficient set is large. DA-based code generation including all VHDL code and test benches is done in a few seconds using the vendor's FIR compilers [95].

Let us have a look at the FIR filter generation of an F6 filter from Goodman and Carey [89] that we had discussed before; see Example 3.5, p. 199. But this time we use the Altera FIR compiler [95] to build the filter. The Altera FIR compiler MegaCore function generates FIR filters optimized for Altera devices. Stratix, Arria and Cyclone IV devices are supported but no mature devices from the APEX or Flex family. You can use the IP toolbench MegaWizard design environment to specify a variety of filter architectures, including fixed-coefficient, multicycle variable, and multirate filters. The FIR compiler includes a coefficient generator, but can also load and use predefined (for instance computed via MATLAB) coefficients from a file.

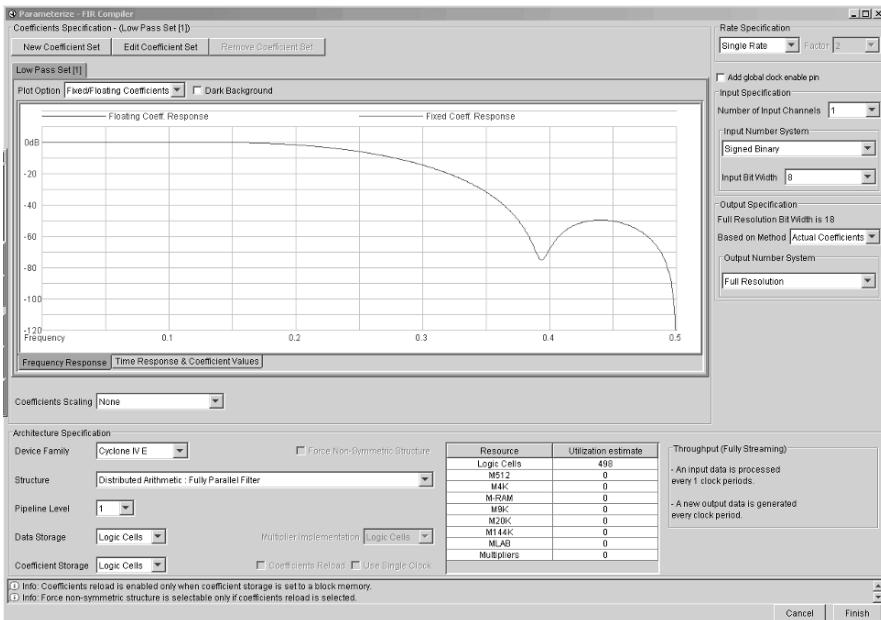


Fig. 3.19. IP parameterization of FIR core according to the F6 Example 3.5, p. 199

Example 3.10: F6 Half-Band Filter IP Generation

To start the Altera FIR compiler we select the **MegaWizard Plug-In Manager** under the **Tools** menu and the library selection window (see Fig. 1.22, p. 43) will pop up. The FIR compiler can be found under **DSP→Filters**. You need to specify a design name for the core and then proceed to the **ToolBench**. We first parameterize the filter and, since we want to use the F6 coefficients, we select **Edit Coefficient Set** and load the coefficient filter by selecting **Imported Coefficient Set**. The coefficient file is a simple text file with each line listing a single coefficient, starting with the first coefficient in the first line. The coefficients can be integer or floating-point numbers, which will then be quantized by the tool since only integer-coefficient filters can be generated with the FIR compiler. The coefficients are shown in the impulse response window as shown in Fig. 3.18b and can be modified if needed. After loading the coefficients we can then select the **Structure** to be fully parallel, fully serial, multi-bit serial, or multi-cycle. We select **Distributed Arithmetic: Fully Parallel Filter**. We set the input coefficient width to 8 bits and let the tool compute the output bit width based on the method **Actual Coefficients**. We select **Coefficient Scaling** as **None** since our integer coefficients should not be further quantized. The transfer function in integer and floating-point should therefore be seen as matching lines; see Fig. 3.19. The FIR compiler reports an estimated size of 498 LEs. In Step 2: **Set Up Simulation** we check all option and let the tool generate all possible simulation files. We proceed with step 3 and the generation of the VHDL code and all supporting files follows. These files are listed in Table 3.6. We see that not only are the VHDL files generated along with their component files, but MODELSTM simulation (RTL and gate-level) scripts, MATLAB (bit accurate)

and Quartus II (cycle accurate) test vectors are also provided to enable an easy verification path. We then compile the HDL code of the filter to enable a timing simulation and provide precise resource data. The MODELSIM simulation of the F6 filter is shown in Fig. 3.20. The simulation first tests the impulse response and then some random test data (not shown here). We see that the signals are not in our usual order or data types, and as a minimum you may want to change the data type from **Binary** to **Decimal**. The wave window shows two most important signals first: **ast_sink_data** is the filter input and **ast_source_data** is the filter output. The **sink** and **source** names are used as the “Avalon Streaming Interface” sees the filter core. The impulse test at 150 ns produces the filter coefficient starting around 300 ns. Note also that several additional control signals have been synthesized, although we did not ask for them.

The design from Example 3.10 requires 490 LEs and runs at 355.37 MHz. The LE count is slightly lower than the estimation of 498 LEs. The overall cost metric measured as the quotient LEs/Fmax is 1.4 and is better than RAG without pipelining, since the DA is fully pipelined, as you can see from the large initial delay of the impulse response. For an appropriate comparison we should compare the DA-based design with the fully pipelined RAG design. The cost of the DA design is higher than the fully pipelined RAG design; see Table 3.5, p. 204. However, the registered performance of the DA-based IP core is slightly higher than the fully pipelined RAG design.

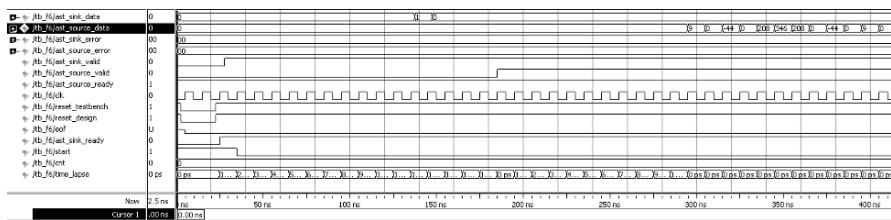


Fig. 3.20. FIR core timing simulation result

3.4.5 Comparison of DA- and RAG-Based FIR Filters

In the last section we followed a detailed case study of the F6 half-band RAG- and DA-based FIR filter designs. The question now would be whether the results were just one single (atypical) example or whether the results in terms of speed/size/cost are typical. In order to answer this question a set of larger filters has been designed using VHDL for fully pipelined RAG optimized via a genetic algorithm [96] and should be compared with the synthesis data using the FIR core compiler from Xilinx that implements a fully parallel DA filter and common sub-expression (CSE) method (see Exercises 3.4 and

Table 3.6. IP files generation for FIR core

File	Description
f6.vhd	A MegaCore function variation file, which defines a top-level VHDL description of the custom MegaCore function
f6.cmp	A VHDL component declaration for the MegaCore function variation
f6.bsf	Quartus II symbol file to be used in the Quartus II block diagram editor
f6_ast.vhd	Avalon streaming interface wrapper
f6_constraints.tcl	This file contains the necessary constraints to achieve FIR filter size and speed
f6_nativelink.tcl	NativeLink simulation test bench
f6_mlab.m	This file provides a MATLAB simulation model for the customized FIR filter
f6_model.m	MATLAB M-file describing a bit-accurate model
f6_vec	This file provides simulation test vectors to be used simulating the customized FIR filter with the Quartus II software
tb_f6.vhd	This file provides a VHDL test bench for the customized FIR filter
f6_msim.tcl	Test bench script file
f6_input.txt	Simulation data for MATLAB
f6_coef_int.txt	Filter coefficient for MATLAB
f6_param.txt	Output parameter data for IP
f6_silent_param.txt	Input parameter data for IP
f6.vho	VHDL IP functional simulation model
f6.qip	Quartus project information
f6.html	The MegaCore function report file

3.5, p. 220) from the literature [97, 98]. Since many of the previous results in the literature have been reported for the Xilinx ISE tools and Virtex 4 devices, we show in Table 3.7 the synthesis result for these tools and devices. The ISE FIR Core Generator 5.0 for the parallel Distributed Arithmetic (PDA) was used. For the CSE comparison the Verilog code and benchmarks posted at http://cseweb.ucsd.edu/~kastner/research/fir_benchmarks was used [98]. As the device the Virtex 4 XC4VSX25-10FF680 was chosen which is large enough to host the largest filter. Table 3.7 shows the results

Table 3.7. Size, speed, and cost comparison of CSE, DA and RAG algorithm for a Xilinx XC4VSX25-10FF680 from the Virtex 4 family [96]

Filter length	CSE			DA			RAG-based		
	LEs	Fmax (MHz)	Cost $\frac{\text{LEs}}{\text{Fmax}}$	LEs	Fmax (MHz)	Cost $\frac{\text{LEs}}{\text{Fmax}}$	LEs	Fmax (MHz)	Cost $\frac{\text{LEs}}{\text{Fmax}}$
6	265	328.7	0.81	368	313.4	1.17	169	323.7	0.52
10	455	285.0	1.60	506	250.4	2.02	246	301.8	0.81
13	385	322.6	1.19	699	305.1	2.29	355	303.0	1.17
20	373	386.5	0.96	1013	260.4	3.89	442	292.4	1.51
28	1245	264.9	4.70	1531	284.5	5.38	628	256.0	2.45
41	1804	257.8	7.00	2135	256.0	8.34	893	262.6	3.40
61	2687	236.5	11.4	3159	271.4	11.6	1267	242.4	5.23
119	5058	235.2	21.5	5991	312.3	19.2	2308	235.1	9.81
151	6468	222.8	29.0	7591	257.1	29.5	2931	221.0	13.3
Mean	2082	282.2	8.68	2554	279.0	9.27	1026	270.9	4.18
Gain	RAG/CSE			RAG/DA					
	-103%	4%	-108%	-149%	3%	-122%			

for 16-bit input data that have a minimum adder depth, i.e., three pipeline stages to be fully pipelined.

It can be seen from Table 3.7 that:

- Pipelined RAG filters enjoy size reductions averaging 103% compared with CSE-based and 149% compared with DA-based designs.
- The register performance of the CSE-based and DA-based FIR filters is on average 4% and 3% higher than pipelined RAG designs.
- The overall cost, measured as LEs/Fmax, is on average 122% better for RAG-based compared with DA-based designs and 108% better than the CSE approach.

Exercises

Note: If you have no prior experience with the Quartus II software, refer to the case study found in Sect. 1.4.3, p. 32. If not otherwise noted use the EP4CE115F29C7 from the Cyclone IV E family for the Quartus II synthesis evaluations.

3.1: A filter has the following specification: sampling frequency 2 kHz; passband 0–0.4 kHz, stopband 0.5–1 kHz; passband ripple 3 dB, and stopband ripple 48 dB. Use the MATLAB software and the “Interactive Lowpass Filter Design” demo or the `fdatool` from the Signal Processing Toolbox for the filter design to:

- (a1) Design a direct filter with a Kaiser window.
 (a2) Determine the filter length and the absolute ripple in the passband.

- (b1) Design an equiripple filter (use the functions `remex` or `firpm`).
 (b2) Determine the filter length and the absolute ripple in the passband.

3.2: (a) Compute the RAG for a length-11 half-band filter F5 that has the nonzero coefficients $f[0] = 256, f[\pm 1] = 150, f[\pm 3] = -25, f[\pm 5] = 3$.

- (b) What is the minimum output bit width of the filter, if the input bit width is 8 bits?

(c1) Write and compile (with the Quartus II compiler) the HDL code for the filter.

(c2) Simulate the filter with impulse and step responses.

(d) Write the VHDL code for the filter in distributed arithmetic, using the state machine approach with the table realized as `LPM_ROM`.

3.3: (a) Compute the RAG for length-11 half-band filter F7 that has the nonzero coefficients $f[0] = 512, f[\pm 1] = 302, f[\pm 3] = -53, f[\pm 5] = 7$.

- (b) What is the minimum output bit width of the filter, if the input bit width is 8 bits?

(c1) Write and compile (with the Quartus II compiler) the VHDL code for the filter.

(c2) Simulate the filter with impulse and step response.

3.4: Hartley [97] has introduced a concept to implement constant coefficient filters, by exploiting common subexpressions across coefficients. For instance, the filter

$$y[n] = \sum_{k=0}^{L-1} a[k]x[n-k], \quad (3.19)$$

with three coefficients $a[k] = \{480, -302, 31\}$. The CSD code of these three coefficients is given by

	512	256	128	64	32	16	8	4	2	1
480 :	1	0	0	0	-1	0	0	0	0	0
-302 :	0	-1	0	-1	0	1	0	0	1	0
31 :	0	0	0	0	1	0	0	0	0	-1

From the table we note that the pattern $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ can be found four times. If we therefore build the temporary variable $h[n] = 2x[n] - x[n-1]$, we can compute the filter output with

$$y[n] = 256h[n] - 16h[n] - 32h[n-1] + h[n-1]. \quad (3.20)$$

(a) Verify (3.20) by substituting $h[n] = 2x[n] - x[n-1]$.

(b) How many adders are required to yield the direct CSD implementation of (3.19) and the implementation with subexpression sharing?

(c1) Implement the filter with subexpression sharing with Quartus II for 8-bit inputs.

(c2) Simulate the impulse response of the filter.

(c3) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks).

3.5: Use the subexpression method from Exercise 3.4 to implement a 4-tap filter with the coefficients $a[k] = \{-1406, -1109, -894, 2072\}$.

(a) Find the CSD code and the subexpression representation for the most frequent pattern.

- (b) Substitute for the subexpression a 2 or -2 , respectively. Apply the subexpression sharing one more time to the reduced set.
 (c) Determine the temporary equations and check by substitution back into (3.19).
 (d) How many adders are required to yield the direct CSD implementation of (3.19) and the implementation with subexpression sharing?
 (e1) Implement the filter with subexpression sharing with Quartus II for 8-bit inputs.
 (e2) Simulate the impulse response of the filter.
 (e3) Determine the registered performance using the **TimeQuest** slow 85C model and the used resources (LEs, multipliers, and M9Ks).

3.6: (a1) Use the program **dagen.exe** to compile a DA table for the coefficients $\{20, 24, 21, 100, 13, 11, 19, 7\}$ using multiple CASE statements.

Synthesize the design for maximum speed and determine the resources (LEs, multipliers, and M9Ks) and registered performance using the **TimeQuest** slow 85C model.

- (a2) Simulate the design using power-of-two $2^k; 0 \leq k \leq 7$ input values.
 (b) Use the partitioning technique to implement the same table using two sets, namely $\{20, 24, 21, 100\}$ and $\{13, 11, 19, 7\}$, and an additional adder. Synthesize the design for maximum speed and determine the size and registered performance using the **TimeQuest** slow 85C model.
 (b2) Simulate the design using power-of-two $2^k; 0 \leq k \leq 7$ input values.
 (c) Compare the designs from (a) and (b).

3.7: Implement 8-bit input/output improved 4-tap $\{-1, 3.75, 3.75, -1\}$ filter designs according to the listing in Table 3.3, p. 195. For each filter write the HDL code and determine the resources (LEs, multipliers, and M9Ks) and registered performance using the **TimeQuest** slow 85C model.

- (a) Synthesize **fir_sym.vhd** as the filter using symmetry.
 (b) Synthesize **fir_csd.vhd** as the filter using CSD coding.
 (c) Synthesize **fir_tree.vhd** as the filter using an adder tree.
 (d) Synthesize **fir_csd_sym.vhd** as the filter using CSD coding and symmetry.
 (e) Synthesize **fir_csd_sym_tree.vhd** as the filter using all three improvements.

3.8: (a) Write a short MATLAB program that plots the

- (a1) impulse response,
 (a2) frequency response, and
 (a2) the pole/zero plot for the half-band filter F3; see Table 5.3, p. 339.

Hint: Use the MATLAB functions: **filter**, **stem**, **freqz**, **zplane**.

- (b) What is the bit growth of the F3 filter? What is the total required output bit width for an 8-bit input?
 (c) Use the **csd.exe** program from the CD to determine the CSD code for the coefficients.
 (d) Use the **ragopt.exe** program from the CD to determine the reduced adder graph (RAG) of the filter coefficients.

3.9: Repeat Exercise 3.8 for the CFIR filter of the GC4114 communication IC. Try the WWW to download a datasheet if possible. The 31 filter coefficients are: $-23, -3, 103, 137, -21, -230, -387, -235, 802, 1851, 81, -4372, -4774, 5134, 20605, 28216, 20605, 5134, -4774, -4372, 81, 1851, 802, -235, -387, -230, -21, 137, 103, -3, -23$.

(Hint: Use 25, 69, and 839 as NOFs)

3.10: Download the datasheet for the GC4114 from the WWW. Use the results from Exercise 3.9.

- (a) Design the 31-tap symmetric CFIR compensation filter as CSD FIR filter in

transposed form (see Fig. 3.3, p. 181) for 8-bit input and an asynchronous reset. Try to match the simulation shown in Fig. 3.21.

- (b) For the device EP4CE115F29C7 from the Cyclone IV E family determine the resources (LEs, multipliers, and M9Ks) and the registered performance using the TimeQuest slow 85C model.

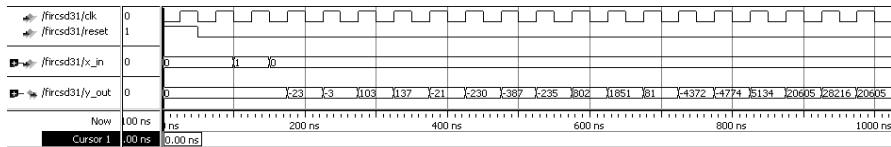


Fig. 3.21. Test bench for the CSD FIR filter in Exercise 3.10

- 3.11:** Download the datasheet for the GC4114 from the WWW. Use the results from Exercise 3.9.

(a) Design the 31-tap symmetric CFIR compensation filter using distributed arithmetic. Use the `dagen.exe` program from the CD to generate the HDL code for the coefficients. Note you should use always groups of four coefficients each and add the results in an adder tree.

(b) Design the DA FIR filter in the full parallel form (see Fig. 3.16, p. 212) for 8-bit input and an asynchronous reset. Take advantage of the coefficient symmetry. Try to match the simulation shown in Fig. 3.22.

(c) For the device EP4CE115F29C7 from the Cyclone IV E family determine the resources (LEs, multipliers, and M9Ks) and the registered performance **F_{max}** using the **TimeQuest** slow 85C model.

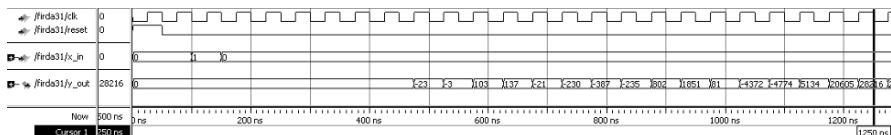


Fig. 3.22. Test bench for the DA-based FIR filter in Exercise 3.11

- 3.12:** Repeat Exercise 3.8 for the half-band filter F4; see Table 5.3, p. 339.

- 3.13:** Repeat Exercise 3.8 for the half-band filter F5; see Table 5.3, p. 339.

- 3.14:** Use the results from Exercise 3.13 and report the HDL code, resources (LEs, multipliers, and M9Ks) and registered performance F_{max} using the TimeQuest slow 85C model for the HDL design of the F5 half-band HDL FIR filter as:

- (a) an RAG filter without pipelining
 (b) a fully pipelined RAG filter
 (c) a DA fully pipelined filter using an FIR core generator

- 3.15:** Repeat Exercise 3.8 for the half-band filter F6; see Table 5.3, p. 339.

3.16: Use the results from Exercise 3.15 and report the HDL code, resources (LEs, multipliers, and M9Ks) and registered performance F_{max} using the TimeQuest slow 85C model for the HDL design of the F6 half-band HDL FIR filter as:

- (a) an RAG filter without pipelining
- (b) a fully pipelined RAG filter
- (c) a DA fully pipelined filter using an FIR core generator

3.17: Repeat Exercise 3.8 for the half-band filter F7; see Table 5.3, p. 339.

3.18: Repeat Exercise 3.8 for the half-band filter F8; see Table 5.3, p. 339.

3.19: Use the results from Exercise 3.18 and report the HDL code, resources (LEs, multipliers, and M9Ks) and registered performance F_{max} using the TimeQuest slow 85C model for the HDL design of the F8 half-band HDL FIR filter as:

- (a) an RAG filter without pipelining
- (b) a fully pipelined RAG filter
- (c) a DA fully pipelined filter using an FIR core generator

3.20: FIR features design. In this problem we want to compare the influence of additional features like reset and enable for different device families. Use the results from Exercise 3.18 for the CSD code. For all following HDL F8 CSD designs with 8-bit input determine the resources (LEs, multipliers, and M4Ks/M9Ks) and registered performance F_{max} using the TimeQuest slow 85C model. As the device use the EP4CE115F29C7 from the Cyclone IV E family and the EP2C35F672C6 from the Cyclone II family.

- (a) Design the F8 CSD FIR filter in direct form (see Fig. 3.1, p. 180).
- (b) Design the F8 CSD FIR filter in transposed form (see Fig. 3.3, p. 181).
- (c) Add a synchronous reset to the transposed FIR from (b).
- (d) Add an asynchronous reset to the transposed FIR from (b).
- (e) Add a synchronous reset and enable to the transposed FIR from (b).
- (f) Add an asynchronous reset and enable to the transposed FIR from (b).
- (g) Tabulate your resources (LEs, multipliers, and M4Ks/M9Ks) and registered performance F_{max} using the TimeQuest slow 85C model results from (a)-(g). What conclusions can be drawn for Cyclone II and IV devices from the measurements?

3.21: Repeat Exercise 3.8 for the half-band filter F9; see Table 5.3, p. 339.

3.22: Use the results from Exercise 3.21 and report the HDL code, resources (LEs, multipliers, and M9Ks) and registered performance F_{max} using the TimeQuest slow 85C model for the HDL design of the F9 half-band HDL FIR filter as:

- (a) an RAG filter without pipelining
- (b) a fully pipelined RAG filter
- (c) a DA fully pipelined filter using an FIR core generator

3.23: Repeat Exercise 3.8 for the Samueli filter S1 [93]. The 25 filter coefficients are: 1, 3, -1, -8, -7, 10, 20, -1, -40, -34, 56, 184, 246, 184, 56, -34, -40, -1, 20, 10, -7, -8, -1, 3, 1.

3.24: Use the results from Exercise 3.23 and report the HDL code, resources (LEs, multipliers, and M9Ks) and registered performance F_{max} using the TimeQuest slow 85C model for the HDL design of the Samueli filter S1 FIR filter as:

- (a) an RAG filter without pipelining
- (b) a fully pipelined RAG filter
- (c) a DA fully pipelined filter using an FIR core generator

3.25: Repeat Exercise 3.8 for the Samueli filter S2 [93]. The 60 filter coefficients are: 31, 28, 29, 22, 8, -17, -59, -116, -188, -268, -352, -432, -500, -532, -529, -464, -336, -129, 158, 526, 964, 1472, 2008, 2576, 3136, 3648, 4110, 4478, 4737, 4868, 4868, 4737, 4478, 4110, 3648, 3136, 2576, 2008, 1472, 964, 526, 158, -129, -336, -464, -529, -532, -500, -432, -352, -268, -188, -116, -59, -17, 8, 22, 29, 28, 31.

3.26: Use the results from Exercise 3.25 and report the HDL code, resources (LEs, multipliers, and M9Ks) and registered performance F_{max} using the TimeQuest slow 85C model for the HDL design of the Samueli filter S2 FIR filter as:

- (a) an RAG filter without pipelining.
- (b) a fully pipelined RAG filter.
- (c) a DA fully pipelined filter using an FIR core generator.

3.27: Repeat Exercise 3.8 for the Lim and Parker L2 filter [94]. The 63 filter coefficients are: 3, 6, 8, 7, 1, -9, -19, -24, -20, -5, 15, 31, 33, 16, -15, -46, -59, -42, 4, 61, 99, 92, 29, -71, -164, -195, -119, 74, 351, 642, 862, 944, 862, 642, 351, 74, -119, -195, -164, -71, 29, 92, 99, 61, 4, -42, -59, -46, -15, 16, 33, 31, 15, -5, -20, -24, -19, -9, 1, 7, 8, 6, 3.

3.28: Use the results from Exercise 3.27 and report the HDL code, resources (LEs, multipliers, and M9Ks) and registered performance F_{max} using the TimeQuest slow 85C model for the HDL design of the Lim and Parker L2 filter FIR filter as:

- (a) an RAG filter without pipelining
- (b) a fully pipelined RAG filter
- (c) a DA fully pipelined filter using an FIR core generator

3.29: Repeat Exercise 3.8 for the Lim and Parker L3 filter [94]. The 36 filter coefficients are: 10, 1, -8, -14, -14, -3, 10, 20, 24, 9, -18, -40, -48, -20, 36, 120, 192, 240, 240, 192, 120, 36, -20, -48, -40, -18, 9, 24, 20, 10, -3, -14, -14, -8, 1, 10.

3.30: Use the results from Exercise 3.29 and report the HDL code, resources (LEs, multipliers, and M9Ks) and registered performance F_{max} using the TimeQuest slow 85C model for the HDL design of the Lim and Parker filter L3 FIR filter as:

- (a) an RAG filter without pipelining
- (b) a fully pipelined RAG filter
- (c) a DA fully pipelined filter using an FIR core generator

4. Infinite Impulse Response (IIR) Digital Filters

Introduction

In Chap. 3 we introduced the FIR filter. The most important properties that make the FIR attractive (+) or unattractive (−) for selective applications include:

- + FIR linear-phase performance is easily achieved.
- + Multiband filters are possible.
- + The Kaiser window method allows iterative-free design.
- + FIRs have a simple structure for decimators and interpolators (see Chap. 5).
- + Nonrecursive filters are always stable and have no limit cycles.
- + It is easy to get high-speed, pipelined designs.
- + FIRs typically have low coefficient and arithmetic roundoff error budgets, and well-defined quantization noise.
- Recursive FIR filters may be unstable because of imperfect pole/zero annihilation.
- The sophisticated Parks–McClellan algorithms must be available for minimax filter design.
- High filter length requires high implementation effort.

Compared to an FIR filter, an IIR filter can often be much more efficient in terms of attaining certain performance characteristics with a given filter order. This is because the IIR filter incorporates feedback and is capable of realizing both zeros and poles of a system transfer function, whereas the FIR filter is an all-zero filter. In this chapter, the fundamentals of IIR filter design will be developed. The traditional approach to the design of IIR filters involves the transformation of an analog filter, with defined feedback specifications, into the digital domain. This is a reasonable approach, mainly because the art of designing analog filters is highly advanced, and many standard tables are available, i.e., [99]. We will review the four most important classes of these analog prototype filters in this chapter, namely Butterworth, Chebyshev I and II, and elliptic filters.

The IIR will be shown to overcome many of the deficiencies of the FIR, but to have some less desirable properties as well. The general desired (+) and undesired (−) properties of an IIR filter are:

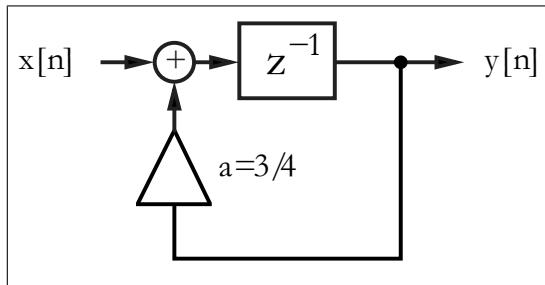


Fig. 4.1. First order IIR filter used as lossy integrator

- + Standard design using an analog prototype filter is well understood.
- + Highly selective filters can be realized with low-order designs that can run at high speeds.
- + Design using tables and a pocket calculator is possible.
- + For the same tolerance scheme, filters are short, compared with FIR filters.
- + Closed-loop design algorithms can be used.
 - Nonlinear-phase response is typical, i.e., it is difficult to get linear-phase response. (Using an allpass filter for phase compensation results in twice the complexity.)
 - Limit cycles may occur for integer implementation.
 - Multiband design is difficult; only low, high, or bandpass filters are designed.
 - Feedback can introduce instabilities. (Most often, the mirror pole to the unit circle can be used to produce the same magnitude response, and the filter will be stable.)
 - It is more difficult to get high-speed, pipelined designs

To demonstrate the possible benefits of using IIR filters, we will discuss a first order IIR filter example.

Example 4.1: Lossy Integrator I

One of the basic tasks of a filter may be to smooth a noisy signal. Assume that a signal $x[n]$ is received in the presence of wideband zero-mean random noise. Mathematically, an integrator could be used to suppress the effects of the noise. If the average value of the input signal is to be preserved over a finite time interval, a *lossy* integrator is often used to process the signal with additive noise. Figure 4.1 displays a simple first order lossy integrator that satisfies the discrete-time difference equation:

$$y[n + 1] = \frac{3}{4}y[n] + x[n]. \quad (4.1)$$

As we can see from the impulse response in Fig. 4.2a, the same functionality of the first order lossy integrator can be achieved with a 15-tap FIR filter. The step response to the lossy integrator is shown in Fig. 4.2b.

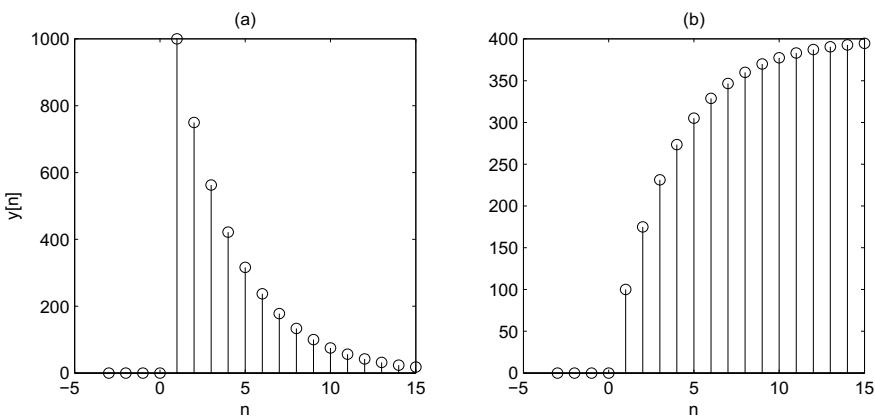


Fig. 4.2. Simulation of lossy integrator with $a = 3/4$. (a) Impulse response for $x[n] = 1000\delta[n]$. (b) Step response for $x[n] = 100\sigma[n]$

The following VHDL code¹ shows a possible implementation of this IIR filter:

```

PACKAGE n_bit_int IS
    -- User defined type
    SUBTYPE S15 IS INTEGER RANGE -2**14 TO 2**14-1;
END n_bit_int;

LIBRARY work;
USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
-----
ENTITY iir IS
    PORT (clk      : IN STD_LOGIC;      -- System clock
          reset   : IN STD_LOGIC;      -- Asynchronous reset
          x_in    : IN  S15;          -- System input
          y_out   : OUT S15);        -- Output result
END iir;
-----
ARCHITECTURE fpga OF iir IS
    SIGNAL x, y : S15;
BEGIN
    PROCESS(reset, clk, x_in, y)
    BEGIN
        -- Use FF for input and recursive part
        IF reset = '1' THEN -- Asynchronous clear
            x <= 0; y <= 0;
        END IF;
        -- Recursive part
        y <= (x_in + y) / 4;
        x <= y;
    END PROCESS;
END;

```

¹ The equivalent Verilog code *iir.v* for this example can be found in Appendix A on page 820. Synthesis results are shown in Appendix B on page 881.

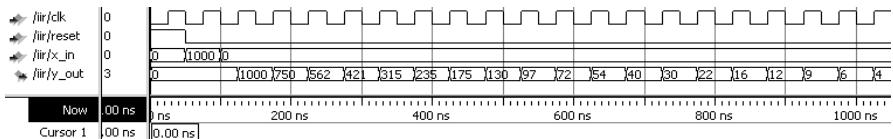


Fig. 4.3. Impulse response for MODELSIM simulation of the lossy integrator

```

ELSIF rising_edge(clk) THEN
    x  <= x_in;
    y  <= x + y / 4 + y / 2;
END IF;
END PROCESS;

y_out <= y;           -- Connect y to output pins

END fpga;

```

Registers have been implemented using a WAIT statement inside a PROCESS block, while the multiplication and addition is implemented using CSD code. The design uses 62 LEs, no embedded multiplier, and has an $F_{max}=147.3$ MHz registered performance using the TimeQuest slow 85C model. The response of the filter to an impulse of amplitude 1000, shown in Fig. 4.3, agrees with the MATLAB simulated results presented in Fig. 4.2a.

4.1

An alternative design approach using a “standard logic vector” data type and LPM_ADD_SUB megafunctions is discussed in Exercise 4.6 (p. 301). This second approach will produce longer VHDL code but will have the benefit of direct control, at the bit level, over the sign extension and multiplier.

4.1 IIR Theory

A nonrecursive filter incorporates, as the name implies, no feedback. The impulse response of such a filter is finite, i.e., it is an FIR filter. A recursive filter, on the other hand has feedback, and is expected, in general, to have an infinite impulse response, i.e., to be an IIR filter. Figure 4.4a shows filters with separate recursive and nonrecursive parts. A *canonical* filter is produced if these recursive and nonrecursive parts are merged together, as shown in Fig. 4.4b. The transfer function of the filter from Fig. 4.4 can be written as:

$$F(z) = \frac{\sum_{l=0}^{L-1} b[l]z^{-l}}{1 - \sum_{l=1}^{L-1} a[l]z^{-l}}. \quad (4.2)$$

The difference equation for such a system yields:

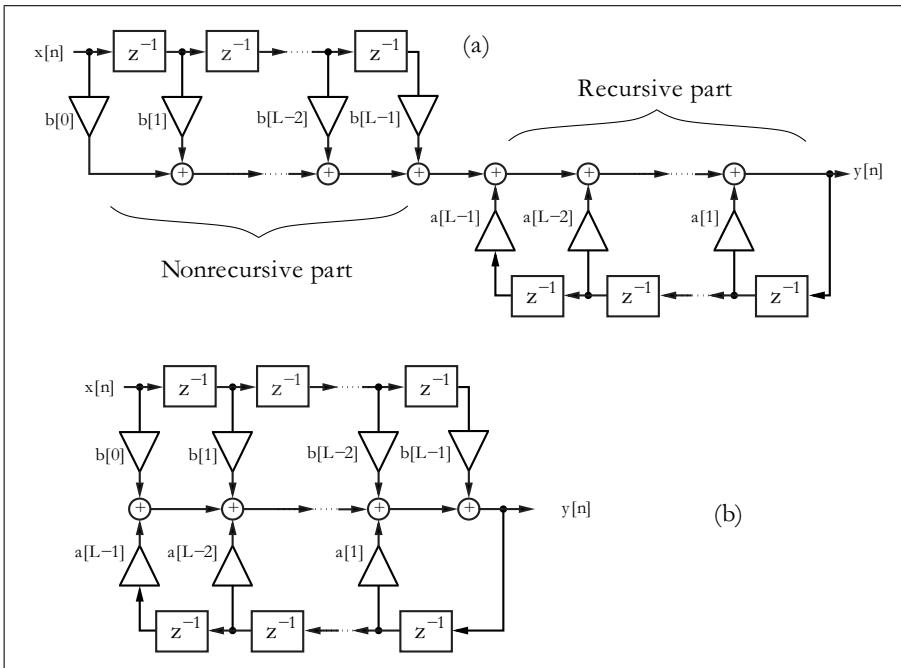


Fig. 4.4. Filter with feedback

$$y[n] = \sum_{l=0}^{L-1} b[l]x[n-l] + \sum_{l=1}^{L-1} a[l]y[n-l]. \quad (4.3)$$

Comparing this with the difference equation for the FIR filter (3.2) on p. 180, we find that the difference equation for recursive systems depends not only on the L previous values of the input sequence $x[n]$, but also on the $L - 1$ previous values of $y[n]$.

If we compute poles and zeros of $F(z)$, we see that the nonrecursive part, i.e., the numerator of $F(z)$, produces the *zeros* p_{0l} , while the denominator of $F(z)$ produces the *poles* $p_{\infty l}$.

For the transfer function, the *pole/zero plot* can be used to look up the most important properties of the filter. If we substitute $z = e^{j\omega T}$ in the z -domain transfer function, we can construct the Fourier transfer function

$$F(\omega) = |F(\omega)|e^{j\theta(\omega)} = \frac{\prod_{l=0}^{L-2} p_{0l} - e^{j\omega T}}{\prod_{l=0}^{L-2} p_{\infty l} - e^{j\omega T}} = \frac{\exp(j \sum_l \beta_l) \prod_{l=0}^{L-2} v_l}{\exp(j \sum_l \alpha_l) \prod_{l=0}^{L-2} u_l} \quad (4.4)$$

by graphical means. This is shown in Fig. 4.5, for a specific amplitude (i.e., gain) and phase value. The gain at a specific frequency ω_0 is the quotient of

the zero vectors v_l and the pole vectors u_l . These vectors start at a specific zero or pole, respectively, and end at the frequency point, $e^{j\omega_0 T}$, of interest. The phase gain for the example from Fig. 4.5 becomes $\theta(\omega_0) = \beta_0 + \beta_1 - \alpha_0$.

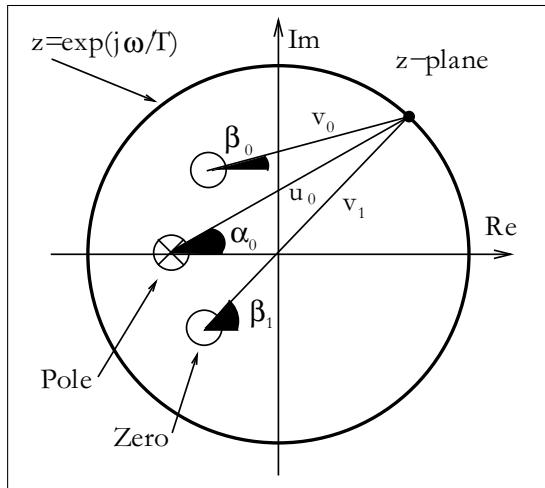


Fig. 4.5. Computation of transfer function using the pole/zero plot. Amplitude gain = $u_0 u_1 / v_0$, phase gain = $\beta_0 + \beta_1 - \alpha_0$

Using the connection between the transfer function in the Fourier domain and the pole/zero plot, we can already deduce several properties:

- 1) A *zero on the unit circle* $p_0 = e^{j\omega_0 T}$ (with no annihilating pole) produces a zero in the transfer function in the Fourier domain at the frequency ω_0 .
- 2) A *pole on the unit circle* $p_\infty = e^{j\omega_0 T}$ (and no annihilating zero) produces an infinite gain in the transfer function in the Fourier domain at the frequency ω_0 .
- 3) A *stable filter* with all poles inside the unit circle can have any type of input signal.
- 4) A *real filter* has single poles and zeros on the real axis, while complex poles and zeros appear always in pairs, i.e., if $a_0 + ja_1$ is a pole or zero, $a_0 - ja_1$ must also be a pole or zero.
- 5) A *linear-phase* (i.e., constant group delay) filter has all poles and zeros symmetric to the unit circle or at $z = 0$.

If we combine observations 3 and 5, we find that, for a stable linear-phase system, all zeros must be symmetric to the unit circle and only poles at $z = 0$ are permitted.

An IIR filter (with poles $z \neq 0$) can therefore be only approximately linear-phase. To achieve this approximation a well-known principle from analog filter design is used: an allpass has a unit gain, and introduces a nonzero

phase gain, which is used to achieve linearization in the frequency range of interest, i.e., the passband.

4.2 IIR Coefficient Computation

In classical IIR design, a digital filter is designed that approximates an ideal filter. The ideal digital filter model specifications are mathematically converted into a set of specifications from an analog filter model using the *bilinear z-transform* given by:

$$s = \frac{z - 1}{z + 1}. \quad (4.5)$$

A classic analog Butterworth, Chebyshev, or elliptic model can be synthesized from these specifications, and is then mapped into a digital IIR using this bilinear *z*-transform.

An analog *Butterworth* filter has a magnitude-squared frequency response given by:

$$|F(\omega)|^2 = \frac{1}{1 + \left(\frac{\omega}{\omega_s}\right)^{2N}}. \quad (4.6)$$

The poles of $|F(\omega)|^2$ are distributed along a circular arc at locations separated by π/N radians. More specifically, the transfer function is N times differentiable at $\omega = 0$. This results in a locally smooth transfer function around 0 Hz. An example of a Butterworth filter model is shown in Fig. 4.6(upper). Note that the tolerance scheme for this design is the same as for the Kaiser window and equiripple design shown in Fig. 3.7 (p. 190).

An analog *Chebyshev* filter of Type I or II is defined in terms of a Chebyshev polynomial $V_N(\omega) = \cos(N \cos(\omega))$, which forces the filter poles to reside on an ellipse. The magnitude-squared frequency response of a Type I filter is represented by:

$$|F(\omega)|^2 = \frac{1}{1 + \varepsilon^2 V_N^2 \left(\frac{\omega}{\omega_s}\right)}. \quad (4.7)$$

An example of a typical Type I magnitude frequency and impulse response is shown in Fig. 4.7(upper). Note the ripple in the passband, and smooth stopband behavior.

The Type II magnitude-squared frequency response is modeled as:

$$|F(\omega)|^2 = \frac{1}{1 + \left(\varepsilon^2 V_N^2 \left(\frac{\omega}{\omega_s}\right)^{-1}\right)}. \quad (4.8)$$

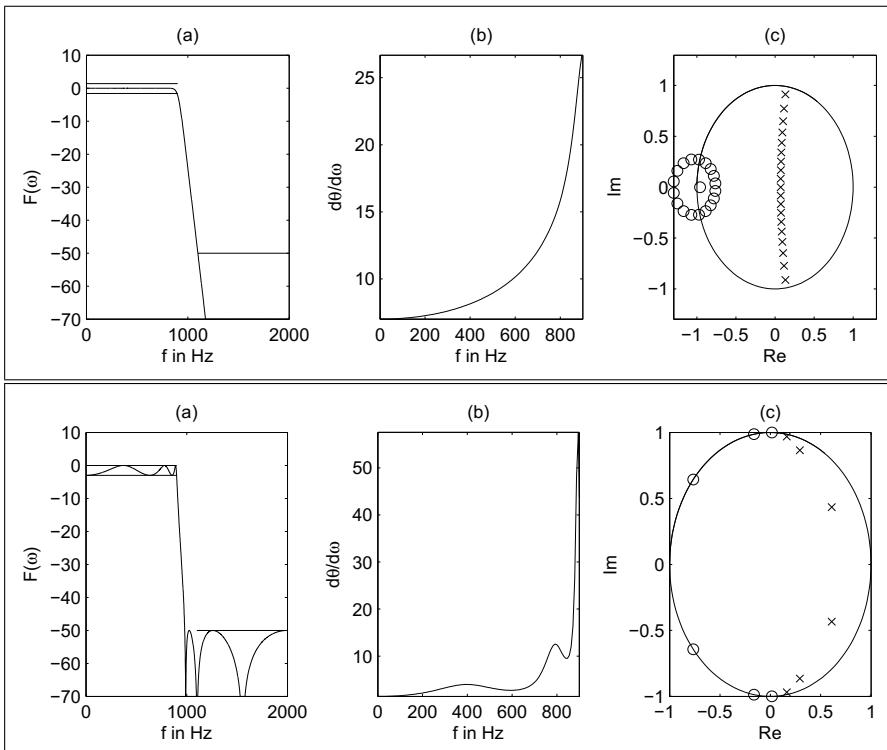


Fig. 4.6. Filter design with MATLAB toolbox. (**upper**) Butterworth filter and (**lower**) elliptic Filter.

(a) Transfer function. (b) Group delay of passband. (c) Pole/zero plot. (\times = pole; \circ = zero)

An example of a typical Type II magnitude frequency and impulse response is shown in Fig. 4.7(lower). Note that in this case a smooth passband results, and the stopband now exhibits ripple behavior.

An analog *elliptic* prototype filter is defined in terms of the solution to the Jacobian elliptic function, $U_N(\omega)$. The magnitude-squared frequency response is modeled as:

$$|F(\omega)|^2 = \frac{1}{1 + \varepsilon^2 U_N^2 \left(\frac{\omega}{\omega_s} \right)^{-1}}. \quad (4.9)$$

The magnitude-squared and impulse response of a typical elliptic filter is shown in Fig. 4.6(lower). Observe that the elliptic filter exhibits ripple in both the passband and stopband.

If we compare the four different IIR filter implementations, we find that a Butterworth filter has order 19, a Chebyshev has order 8, while the elliptic design has order 6, for the same tolerance scheme shown in Fig. 3.8 (p. 191).

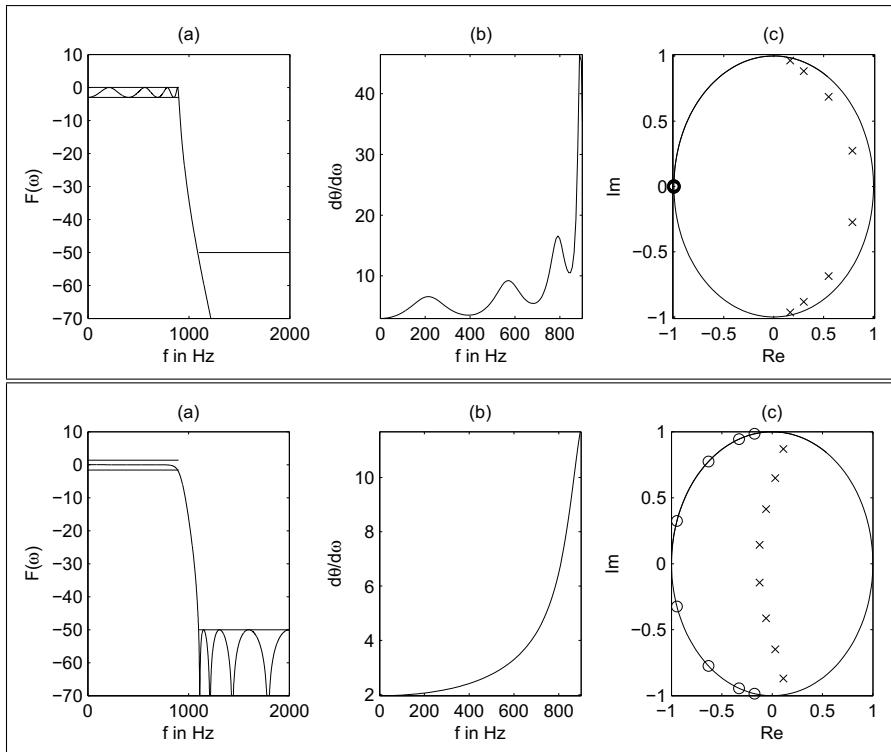


Fig. 4.7. Chebyshev filter design with MATLAB toolbox. Chebyshev I (**upper**) and Chebyshev II (**lower**).

(a) Transfer function. (b) Group delay of passband. (c) Pole/zero plot (\times = pole; \circ = zero)

If we compare Figs. 4.6 and 4.7, we find that for the filter with shorter order the ripple increases, and the group delay becomes highly nonlinear. A good compromise is most often the Chebyshev Type II filter with medium order, a flat passband, and tolerable group delay.

4.2.1 Summary of Important IIR Design Attributes

In the previous section, classic IIR types were presented. Each model provides the designer with tradeoff choices. The attributes of classic IIR types are summarized as follows:

- **Butterworth:** Maximally flat passband, flat stopband, wide transition band
- **Chebyshev I:** Equiripple passband, flat stopband, moderate transition band

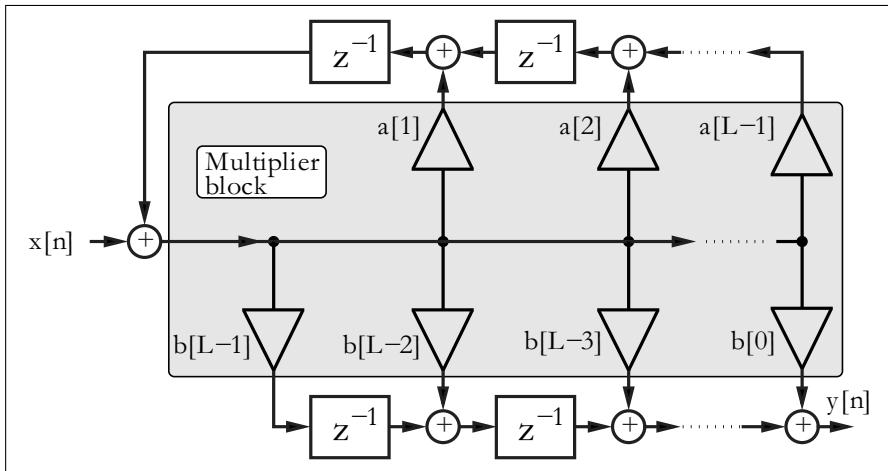


Fig. 4.8. Transposed direct form I IIR filter using multiplier blocks

- **Chebyshev II:** Flat passband, equiripple stopband, moderate transition band
- **Elliptic:** Equiripple passband, equiripple stopband, narrow transition band

For a given set of filter requirement, the following observations generally hold:

- Filter order
 - Lowest: Elliptic
 - Medium: Chebyshev I or II
 - Highest: Butterworth
- Passband characteristics
 - Equiripple: Elliptic, Chebyshev I
 - Flat: Butterworth, Chebyshev II
- Stopband characteristics
 - Equiripple: Elliptic, Chebyshev II
 - Flat: Butterworth, Chebyshev I
- Transition band characteristics
 - Narrowest: Elliptic
 - Medium: Chebyshev I+II
 - Widest: Butterworth

4.3 IIR Filter Implementation

Obtaining an IIR transfer function is generally considered to be a straightforward exercise, especially if design software like MATLAB is used. IIR filters

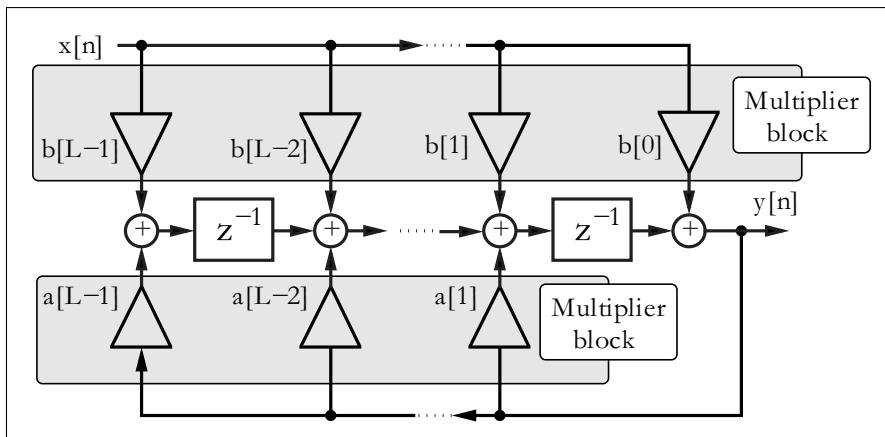


Fig. 4.9. Transposed direct form II IIR filter using multiplier blocks

can be developed in the context of many architectures. The most important structures are summarized as follows:

- Direct I form (see Fig. 4.8)
- Direct II form (see Fig. 4.9)
- Cascade of first- or second order systems (see Fig. 4.10a)
- Parallel implementation of first- or second order systems (see Fig. 4.10b).
- BiQuad implementation of a typical second order section found in basic cascade or parallel designs (see Fig. 4.11)
- Normal [100], i.e., cascade of first- or second order state variable systems (see Fig. 4.10a)
- Parallel normal, i.e., parallel first- or second order state variable systems (see Fig. 4.10b)
- Continued fraction structures
- Lattice filter (after Gray–Markel, see Fig. 4.12)
- Wave digital implementation (after Fettweis [101])
- General state space filter

Each architecture serves a unique purpose. Some of the general selection rules are summarized below:

- Speed
 - High: Direct I & II
 - Low: Wave
- Fixed-point arithmetic roundoff error sensitivity
 - High: Direct I & II
 - Low: Normal, Lattice
- Fixed-point coefficient roundoff error sensitivity

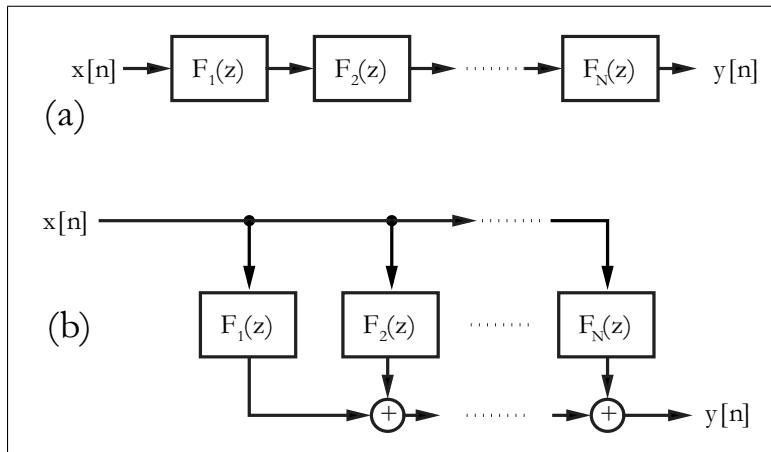


Fig. 4.10. (a) Cascade implementation $F(z) = \prod_{k=1}^N F_k(z)$. (b) Parallel implementation $F(z) = \sum_{k=1}^N F_k(z)$

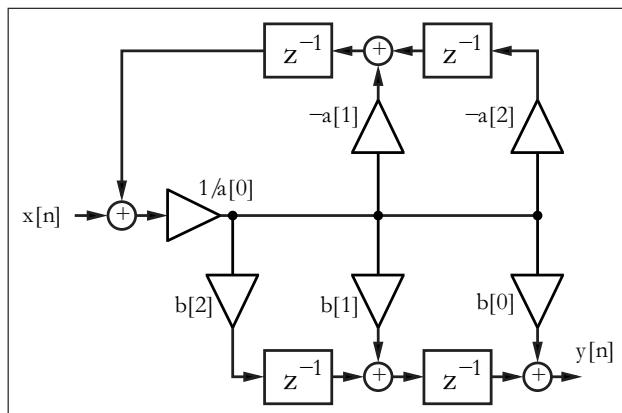


Fig. 4.11. Possible second order section BiQuad with transfer function $F(z) = (b[0] + b[1]z^{-1} + b[2]z^{-2})/(a[0] + a[1]z^{-1} + a[2]z^{-2})$. A BiQuad has two quadratic equations in the transfer function

- High: Direct I & II
- Low: Parallel, Wave
- Special properties
 - Orthogonal weight outputs: Lattice
 - Optimized second order sections: Normal
 - Arbitrary IIR specification: State variable

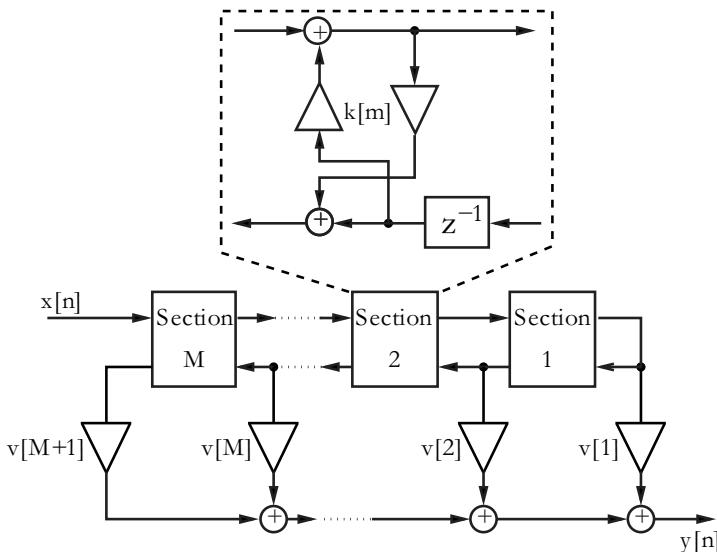


Fig. 4.12. Lattice IIR filter

With the help of software tools like MATLAB, the coefficients can easily be converted from one architecture to another, as demonstrated by the following example.

Example 4.2: Butterworth Second Order System

Assume we wish to design a Butterworth filter (order $N = 10$, passband $F_p = 0.3 \text{ Fs}$) realized by second order systems. We can use the following MATLAB code to generate the coefficients:

```
N=10;Fp=0.3;
[B,A]=butter(N,Fp)
[sos, gain]=tf2sos(B,A)
```

i.e., we first compute the Butterworth coefficient using the function `butter()`, and then convert this filter coefficient using the “transfer function to second order section” function `tf2sos` to compute the BiQuad coefficients. We will get the following results using MATLAB for the second order sections:

$b[0, i]$	$b[1, i]$	$b[2, i]$	$a[0, i]$	$a[1, i]$	$a[2, i]$
1.0000	2.1181	1.1220	1.0000	-0.6534	0.1117
1.0000	2.0703	1.0741	1.0000	-0.6831	0.1622
1.0000	1.9967	1.0004	1.0000	-0.7478	0.2722
1.0000	1.9277	0.9312	1.0000	-0.8598	0.4628
1.0000	1.8872	0.8907	1.0000	-1.0435	0.7753

and the gain is 4.961410^{-5} .

Figure 4.13 shows the transfer function, group delay, and the pole/zero plot of the filter. Note that all zeros are near $z_0 = -1$, which can also be seen

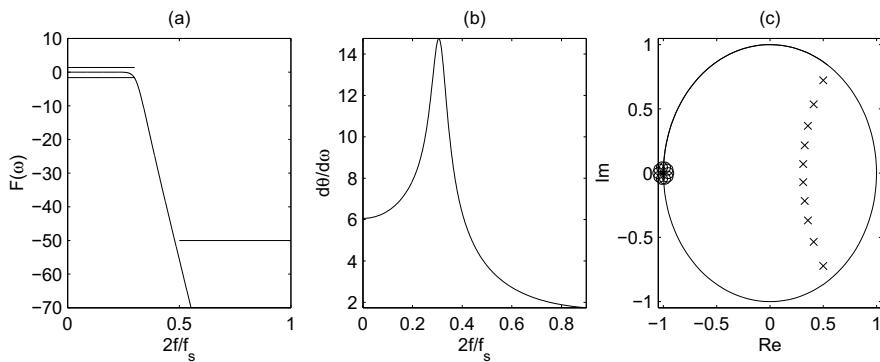


Fig. 4.13. Tenth order Butterworth filter showing (a) magnitude, (b) group delay response, and (c) Pole/zero plot. (\times = pole; \circ = zero)

from the numerator coefficients of the second order systems. Note also the rounding error in $b[1, i] = 2$ and $b[0, i] = b[2, i] = 1$.

4.2

4.3.1 Finite Wordlength Effects

Crochiere and Oppenheim [102] have shown that the coefficient wordlength required for a digital filter is closely related to the coefficient sensitivities. Implementation of the same IIR filter can therefore lead to a wide range of required wordlengths. To illustrate some of the dynamics of this problem, consider an eighth order elliptic filter analyzed by Crochiere and Oppenheim [102]. The resulting eighth order transfer function was implemented with a Wave, Cascade, Parallel, Lattice, Direct I and II, and Continuous Fraction architecture. The estimated coefficient wordlength to meet a specific maximal passband error criterion was conservatively estimated as shown in the second column of Table 4.1. As a result, it can be seen that the Direct form needs more wordlength than the Wave or Parallel structure. This has led to the conclusion that a Wave structure gives the best complexity (MW) in terms of the bit-width (W) multiplier product (M), as can be seen from column six of Table 4.1.

In the context of FIR filters (see Chap. 3), the reduced adder graph (RAG) technique was introduced in order to simplify the design of a block of several multipliers [103, 104]. Dempster and Macleod have evaluated the eighth order elliptic filter from above, in the context of RAG multiplier implementation strategies. A comparison is presented in Table 4.2. The second column displays the multiplier block size. For a Direct II architecture, two multiplier blocks, of size 9 and 7, are required. For a Wave architecture, no two coefficients have the same input, and, as a result, no multiplier blocks can be developed. Instead, eleven individual multipliers must be implemented. The

Table 4.1. Data for eighth order elliptic filter by Crochiere and Oppenheim [102] sorted according the costs $M \times W$

Type	Word-length W	Mults M	Adds	Delays	Cost $M \times W$
Wave	11.35	12	31	10	136
Cascade	11.33	13	16	8	147
Parallel	10.12	18	16	8	182
Lattice	13.97	17	32	8	238
Direct I	20.86	16	16	16	334
Direct II	20.86	16	16	8	334
Cont.-frac	22.61	18	16	8	408

third column displays the number of adders/subtractors B for a canonical signed digit (CSD) design required to implement the multiplier blocks. Column four shows the same result for single-optimized multiplier adder graphs (MAG) [105]. Column five shows the result for the reduced adder graph. Column six shows the overall adder/wordwidth product for a RAG design. Table 4.2 shows that Cascade and Parallel forms give comparable or better results, compared with Wave digital filters, because the multiplier block size is an essential criterion when using the RAG algorithms. Delays have not been considered for the FPGA design, because all the logic cells have an associated flip-flop.

4.3.2 Optimization of the Filter Gain Factor

In general, we derive the IIR integer coefficients from floating-point filter coefficients by first normalizing to the maximum coefficient, and then multiplying with the desired gain factor, i.e., bit-width $2^{\text{round}(W)}$. However, most often it is more efficient to select the gain factor within a range, $2^{[W]} \dots 2^{[W]}$. There will be essentially no change in the transfer function, because the coefficients

Table 4.2. Data for eighth order elliptic filter implemented using CSD, MAG, and RAG strategies [103]

Type	Block size	CSD	MAG	RAG	
		B	B	B	$W(B + A)$
Cascade	$4 \times 3, 2 \times 1$	26	26	24	453
Parallel	$1 \times 9, 4 \times 2, 1 \times 1$	31	30	29	455
Wave	11×1	58	63	22	602
Lattice	$1 \times 9, 8 \times 1$	33	31	29	852
Direct I	1×16	103	83	36	1085
Direct II	$1 \times 9, 1 \times 7$	103	83	41	1189
Cont.-frac	18×1	118	117	88	2351

Table 4.3. Variation of the gain factor to minimize filter complexity of the cascade filter

	CSD	MAG	RAG
Optimal gain	1122	1121	1121
# adders for optimal gain	23	21	18
# adders for gain = 1024	26	26	24
Improvement	12%	19%	25%

must be rounded anyway, after multiplying by the gain factor. If we apply, for instance, this search in the range $2^{\lfloor W \rfloor} \dots 2^{\lceil W \rceil}$ for the cascade filter in the Crochiere and Oppenheim design example from above (gain used in Table 4.2 was $2^{\lfloor 11.33 \rfloor - 1} = 1024$), we get the data reported in Table 4.3.

We note, from the comparison shown in Table 4.3 a substantial improvement in the number of adders required to implement the multiplier. Although the optimal gain factor for MAG and RAG in this case is the same, it can be different.

4.4 Fast IIR Filter

In Chap. 3, FIR filter registered performance was improved using pipelining (see Table 3.3, p. 195). In the case of FIR filters, pipelining can be achieved at essentially no cost. Pipelining IIR filters, however, is more sophisticated and is certainly not free. Simply introducing pipeline registers for all adders will, especially in the feedback path, very likely change the pole locations and therefore the transfer function of the IIR filter. However strategies that do not change the transfer function and still allow a higher throughput have been reported in the literature. The reported methods that look promising to improve IIR filter throughput are:

- Look-ahead interleaving in the time domain [106]
- Clustered look-ahead pole/zero assignment [107, 108]
- Scattered look-ahead pole/zero assignment [106, 109]
- IIR decimation filter design [110]
- Parallel processing [111]
- RNS implementation [44, Sect. 4.2] [55]

The first five methods are based on filter architecture or signal flow techniques, and the last is based on computer arithmetic (see Chap. 2). These techniques will be demonstrated with examples. To simplify the VHDL representation of each case, only a first order IIR filter will be considered, but the same ideas can be applied to higher order IIR filters and can be found in the literature references.

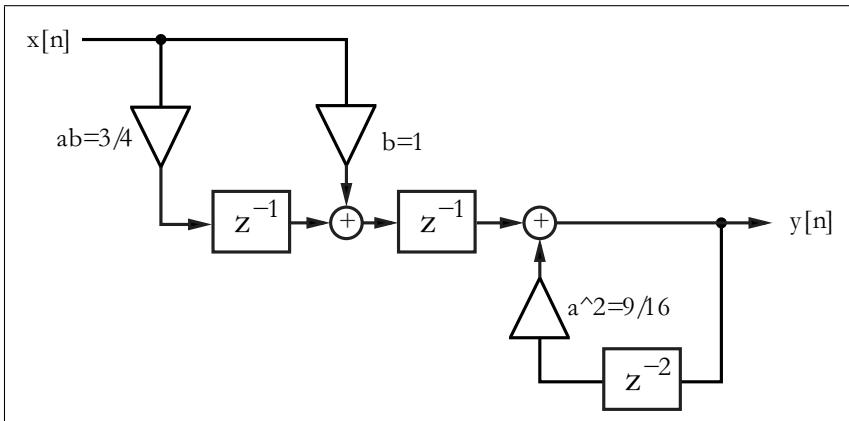


Fig. 4.14. Lossy integrator with look-ahead arithmetic

4.4.1 Time-domain Interleaving

Consider the differential equation of a first order IIR system, namely

$$y[n+1] = ay[n] + bx[n]. \quad (4.10)$$

The output of the first order system, namely $y[n+1]$, can be computed using a look-ahead methodology by substituting $y[n+1]$ into the differential equation for $y[n+2]$. That is

$$y[n+2] = ay[n+1] + bx[n+1] = a^2y[n] + abx[n] + bx[n+1]. \quad (4.11)$$

The equivalent system is shown in Fig. 4.14.

This concept can be generalized by applying the look-ahead transform for $(S - 1)$ steps, resulting in:

$$y[n+S] = a^S y[n] + \underbrace{\sum_{k=0}^{S-1} a^k b x[n+S-1-k]}_{(\eta)}. \quad (4.12)$$

It can be seen that the term (η) defines an FIR filter having coefficients $\{b, ab, a^2b, \dots, a^{S-1}b\}$, that can be pipelined using the pipelining techniques presented in Chap. 3 (i.e., pipelined multiplier and pipelined adder trees). The recursive part of (4.12) can now also be implemented with an S -stage pipelined multiplier for the coefficient a^S . We will demonstrate the look-ahead design with the following example.

Example 4.3: Lossy Integrator II

Consider again the lossy integrator from Example 4.1 (p. 226), but now with look-ahead. Figure 4.14 shows the look-ahead lossy integrator, which is a

combination of a nonrecursive part (i.e., FIR filter for x), and a recursive part with delay 2 and coefficient 9/16.

$$\begin{aligned} y[n+2] &= \frac{3}{4}y[n+1] + x[n+1] = \frac{3}{4}\left(\frac{3}{4}y[n] + x[n]\right) + x[n+1] \\ &= \frac{9}{16}y[n] + \frac{3}{4}x[n] + x[n+1]. \end{aligned} \quad (4.13)$$

$$y[n] = \frac{9}{16}y[n-2] + \frac{3}{4}x[n-2] + x[n-1] \quad (4.14)$$

(4.15)

The VHDL code² shown below, implements the IIR filter in look-ahead form.

```

PACKAGE n_bit_int IS                         -- User defined type
    SUBTYPE S15 IS INTEGER RANGE -2**14 TO 2**14-1;
END n_bit_int;

LIBRARY work;
USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
-----
ENTITY iir_pipe IS
    PORT ( clk      : IN STD_LOGIC; -- System clock
           reset   : IN STD_LOGIC; -- Asynchronous reset
           x_in    : IN S15;       -- System input
           y_out   : OUT S15);    -- System output
END iir_pipe;
-----
ARCHITECTURE fpga OF iir_pipe IS

SIGNAL x, x3, sx, y, y9 : S15;

BEGIN

PROCESS(clk, reset, x_in, x, x3, sx, y, y9)
BEGIN      -- Use FFs for input, output and pipeline stages
    IF reset = '1' THEN -- Asynchronous clear
        x <= 0; x3 <= 0; sx <= 0; y9 <= 0; y <= 0;
    ELSIF rising_edge(clk) THEN
        x    <= x_in;
        x3   <= x / 2 + x / 4;   -- Compute x*3/4
        sx   <= x + x3; -- Sum of x elements = output FIR part
        y9   <= y / 2 + y / 16; -- Compute y*9/16
        y    <= sx + y9;        -- Compute output
    END IF;
END PROCESS;

y_out <= y ;      -- Connect register y to output pins

```

² The equivalent Verilog code *iir_pipe.v* for this example can be found in Appendix A on page 821. Synthesis results are shown in Appendix B on page 881.

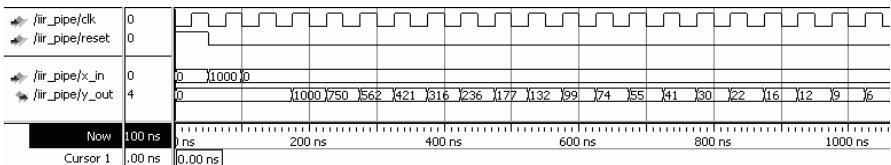


Fig. 4.15. VHDL simulation of impulse response of the look-ahead lossy integrator

END fpga;

The pipelined adder and multiplier in this example are implemented in two steps. In the first stage, $\frac{9}{16}y[n]$ is computed. In the second stage, $x[n + 1] + \frac{3}{4}x[n]$ and $\frac{9}{16}y[n]$ are added. The design uses 123 LEs, no embedded multiplier and Fmax=215.05 MHz registered performance using the TimeQuest slow 85C model. The response of the filter to an impulse of amplitude 1000 is shown in Fig. 4.15.

4.3

Comparing the look-ahead scheme with the 62 LEs and 147.3 MHz solution reported in Example 4.1 (p. 226), we find that look-ahead pipelining requires many more resources, but attains a speed-up of about 30%. The comparison of the two filter's response to the impulse with amplitude 1000 shown in Fig. 4.3 (p. 228) and Fig. 4.15 reveals that the look-ahead scheme has an additional overall delay, and that the quantization effect differs by a ± 2 amount between the two methodologies.

An alternative design approach, using a standard logic vector data type and LPM_ADD_SUB megafunctions, is discussed in Exercise 4.7 (p. 301). The second approach will produce longer VHDL code, but will have the benefit of direct control at the bit level of the sign extension and multiplier.

4.4.2 Clustered and Scattered Look-Ahead Pipelining

Clustered and scattered look-ahead pipelining schemes add self-canceling poles and zeros to the design to facilitate pipelining of the recursive portion of the filter. In the *clustered* method, additional pole/zeros are introduced in such a way that in the denominator of the transfer function the coefficients for $z^{-1}, z^{-2}, \dots, z^{-(S-1)}$ become zero. The following example shows clustering for a second order filter.

Example 4.4: Clustering Method

A second order transfer function is assumed to have a pole at 1/2 and 3/4 and a transfer function given by:

$$F(z) = \frac{1}{1 - 1.25z^{-1} + 0.375z^{-2}} = \frac{1}{(1 - 0.5z^{-1})(1 - 0.75z^{-1})}. \quad (4.16)$$

Adding a canceling pole/zero at $z = -1.25$ results in a new transfer function

$$F(z) = \frac{1 + 1.25z^{-1}}{1 - 1.1875z^{-2} + 0.4688z^{-3}}. \quad (4.17)$$

The recursive part of the filter can now be implemented with an additional pipeline stage.

4.4

The problem with clustering is that the cancelled pole/zero pair may lie outside the unit circle, as is the case in the previous example (i.e., $z_\infty = -1.25$). This introduces instability into the design if the pole/zero annihilating is not perfect. In general, a second order system with poles at r_1, r_2 and with one extra canceling pair, has a pole location at $-(r_1 + r_2)$, which lies outside the unit circle for $|r_1 + r_2| > 1$. Soderstrand et al. [108], have described a stable clustering method, which in general introduces more than one canceling pole/zero pair.

The *scattered look-ahead* approach does not introduce stability problems. It introduces $(S - 1)$ canceling pole/zero pairs located at $z_k = pe^{j\pi k/S}$, for an original filter with a pole located at p . The denominator of the transfer function has, as a result, only zero coefficients associated with the terms z^0, z^S, z^{-2S} , etc.

Example 4.5: Scattered Look-Ahead Method

Consider implementing a second order system having poles located at $z_{\infty 1} = 0.5$ and $z_{\infty 2} = 0.75$ with two additional pipeline stages. A second order transfer function of a filter with poles at $1/2$ and $3/4$ has the transfer function

$$F(z) = \frac{1}{1 - 1.25z^{-1} + 0.375z^{-2}} = \frac{1}{(1 - 0.5z^{-1})(1 - 0.75z^{-1})}. \quad (4.18)$$

Note that in general a pole/zero pair at p and p^* results in a transfer function of

$$(1 - pz^{-1})(1 - p^*z^{-1}) = 1 - (p + p^*)z^{-1} + rr^*z^{-2}$$

and in particular with $p = r \times \exp(j2\pi/3)$ it follows that

$$\begin{aligned} (1 - pz^{-1})(1 - p^*z^{-1}) &= 1 - 2r \cos(2\pi/3)z^{-1} + r^2z^{-2} \\ &= 1 + rz^{-1} + r^2z^{-2}. \end{aligned}$$

The scattered look-ahead introduces two additional pipeline stages by adding pole/zero pairs at $0.5e^{\pm j2\pi/3}$ and $0.75e^{\pm j2\pi/3}$. Adding a canceling pole/zero at this location results in

$$\begin{aligned} F(z) &= \frac{1}{1 - 1.25z^{-1} + 0.375z^{-2}} \\ &\times \frac{(1 + 0.5z^{-1} + 0.25z^{-2})(1 + .75z^{-1} + 0.5625z^{-2})}{(1 + 0.5z^{-1} + 0.25z^{-2})(1 + .75z^{-1} + 0.5625z^{-2})} \\ &= \frac{1 + 1.25z^{-1} + 1.1875z^{-2} + 0.4687z^{-3} + 0.1406z^{-4}}{1 - 0.5469z^{-3} + 0.0527z^{-6}} \\ &= \frac{512 + 640z^{-1} + 608z^{-2} + 240z^{-3} + 72z^{-4}}{512 - 280z^{-3} + 27z^{-6}}, \end{aligned}$$

and the recursive part can be implemented with two additional pipeline stages.

4.5

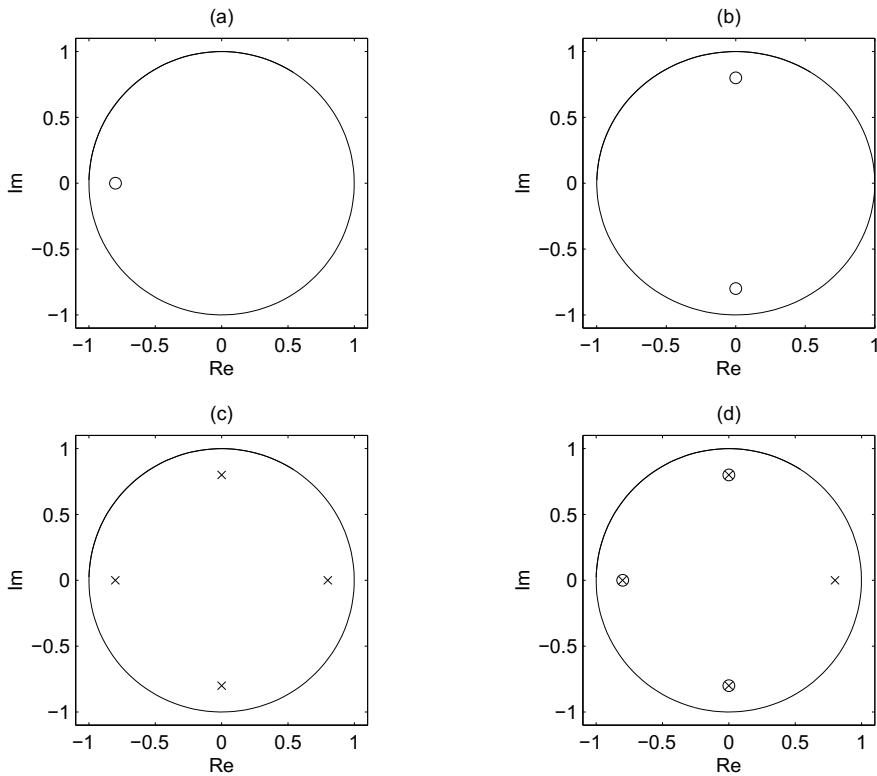


Fig. 4.16. Pole/zero plot for scattered look-ahead first order IIR filter.
(a) $F_1(z) = (1 + az^{-1})$. **(b)** $F_2(z) = 1 + a^2z^{-2}$. **(c)** $F_3(z) = 1/(1 - a^4z^{-4})$.
(d) $F(z) = \prod_k F_k(z) = (1 + az^{-1})(1 + a^2z^{-2})/(1 - a^4z^{-4}) = 1/(1 - az^{-1})$

It is interesting to note that for a first order IIR system, clustered and scattered look-ahead methods result in the same pole/zero canceling pair lying on a circle around the origin with angle differences $2\pi/S$. The nonrecursive part can be realized with a “power-of-two decomposition” according to

$$(1 + az^{-1})(1 + a^2z^{-2})(1 + a^4z^{-4}) \dots \quad (4.19)$$

Figure 4.16 shows such a pole/zero representation for a first order section, which enables an implementation with four pipeline stages in the recursive part.

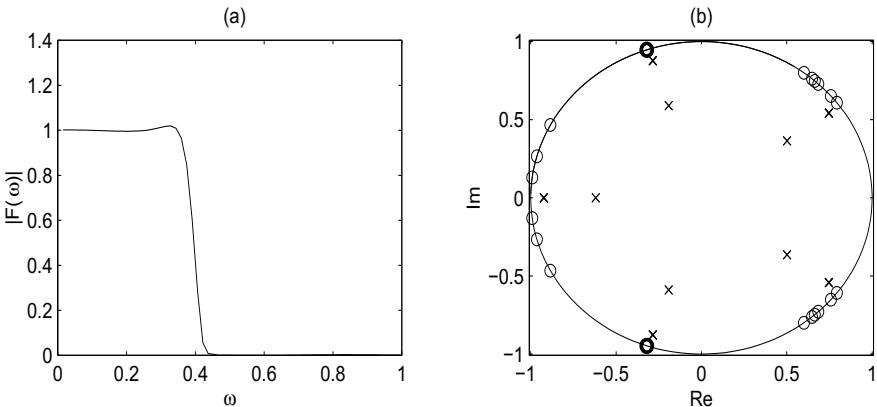


Fig. 4.17. (a) Transfer function, and (b) pole/zero distribution of a 37 order Martinez–Parks IIR filter with $S = 5$

4.4.3 IIR Decimator Design

Martinez and Parks [110] have introduced, in the context of decimation filters (see Chap. 5, p. 314), a filter design algorithm based on the minimax method. The resulting transfer function satisfies

$$F(z) = \frac{\sum_{l=0}^L b[l]z^{-l}}{1 - \sum_{n=0}^{N/S} a[n]z^{-nS}}. \quad (4.20)$$

That is, only every other S coefficient in the denominator is nonzero. In this case, the recursive part (i.e., the denominator) can be pipelined with S stages. It has been found that in the resulting pole/zero distribution, all zeros are on the unit circle, as is usual for an elliptic filter, while the poles lie on circles, whose main axes have a difference in angle of $2\pi/S$, as shown in Fig. 4.17b.

4.4.4 Parallel Processing

In a parallel-processing filter implementation [111], P parallel IIR paths are formed, each running at a $1/P$ input sampling rate. They are combined at the output using a multiplexer, as shown in Fig. 4.18. Because a multiplexer, in general, will be faster than a multiplier and/or adder, the parallel approach will be faster. Furthermore, each path P has a factor of P more time to compute its assigned output.

To illustrate, consider again a first order system and $P = 2$. The look-ahead scheme, as in (4.11)

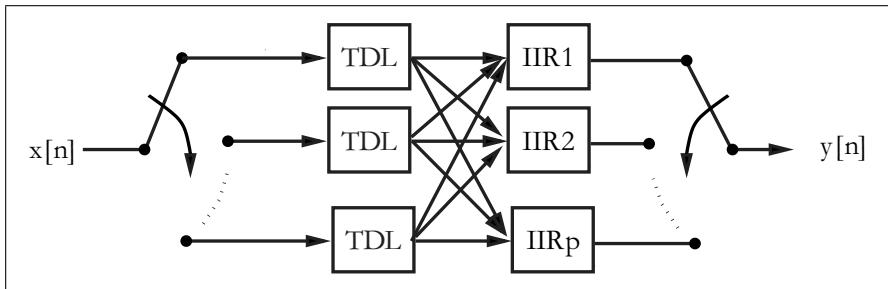


Fig. 4.18. Parallel IIR implementation. The tapped delay lines (TDL) run with a $1/p$ input sampling rate

$$y[n+2] = ay[n+1] + x[n+1] = a^2y[n] + ax[n] + x[n+1] \quad (4.21)$$

is now split into even $n = 2k$ and odd $n = 2k - 1$ output sequences, obtaining

$$y[n+2] = \begin{cases} y[2k+2] = a^2y[2k] + ax[2k] + x[2k+1] \\ y[2k+1] = a^2y[2k-1] + ax[2k-1] + x[2k] \end{cases}, \quad (4.22)$$

where $n, k \in \mathbb{Z}$. The two equations are the basis for the following parallel IIR filter FPGA implementation.

Example 4.6: Lossy Integrator III

Consider implementing a parallel lossy integrator, with $a = 3/4$, as an extension to the methods presented in Examples 4.1 (p. 226) and 4.3 (p. 241). A two-channel parallel lossy integrator, which is a combination of two non-recursive parts (i.e., an FIR filter for x), and two recursive parts with delay 2 and coefficient $9/16$, is shown in Fig. 4.19. The VHDL code³ shown below implements the design.

```
PACKAGE n_bit_int IS           -- User defined type
  SUBTYPE S15 IS INTEGER RANGE -2**14 TO 2**14-1;
```

³ The equivalent Verilog code `iir_par.v` for this example can be found in Appendix A on page 822. Synthesis results are shown in Appendix B on page 881.

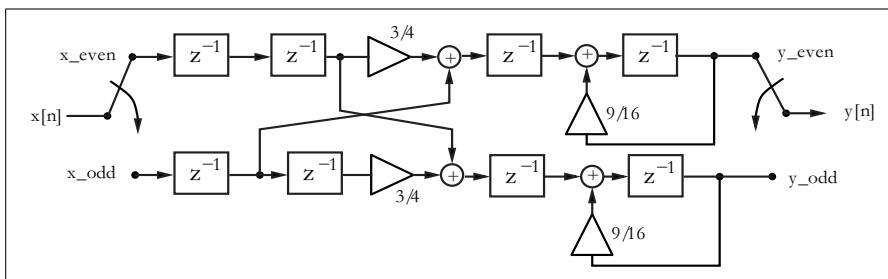


Fig. 4.19. Two-path parallel IIR filter implementation

```

END n_bit_int;

LIBRARY work;
USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
-----
ENTITY iir_par IS          -----> Interface
  PORT ( clk      : IN STD_LOGIC; -- System clock
         reset    : IN STD_LOGIC; -- Asynchronous reset
         x_in     : IN S15;      -- System input
         x_e, x_o : OUT S15;    -- Even/odd input x_in
         y_e, y_o : OUT S15;    -- Even/odd output y_out
         clk2     : OUT STD_LOGIC; -- Clock divided by 2
         y_out    : OUT S15);   -- System output
END iir_par;
-----
ARCHITECTURE fpga OF iir_par IS

  TYPE STATE_TYPE IS (even, odd);
  SIGNAL state           : STATE_TYPE;
  SIGNAL x_even, xd_even : S15 := 0;
  SIGNAL x_odd, xd_odd, x_wait : S15 := 0;
  SIGNAL y_even, y_odd, y_wait, y : S15 := 0;
  SIGNAL sum_x_even, sum_x_odd : S15 := 0;
  SIGNAL clk_div2        : STD_LOGIC := '0';

BEGIN

  Multiplex: PROCESS (reset, clk) --> Split x into even and
  BEGIN                         -- odd samples; recombine y at clk rate
    IF reset = '1' THEN          -- asynchronous reset
      state <= even; x_even <= 0; x_odd <= 0;
      y <= 0; x_wait <= 0; y_wait <= 0;
    ELSIF rising_edge(clk) THEN
      CASE state IS
        WHEN even =>
          x_even <= x_in;
          x_odd <= x_wait;
          clk_div2 <= '1';
          y <= y_wait;
          state <= odd;
        WHEN odd =>
          x_wait <= x_in;
          y <= y_odd;
          y_wait <= y_even;
          clk_div2 <= '0';
          state <= even;
      END CASE;
    END IF;
  END PROCESS Multiplex;

```

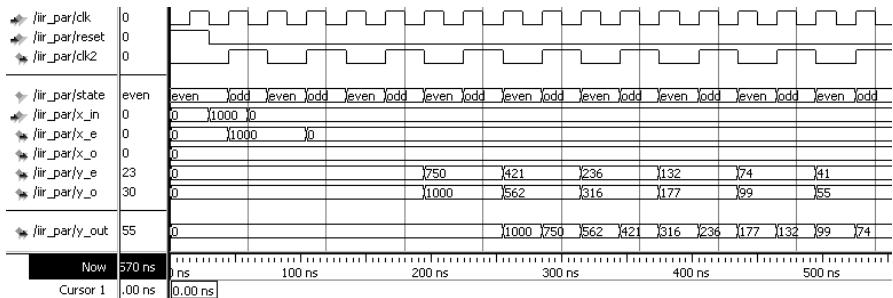


Fig. 4.20. VHDL simulation of the response of the parallel IIR filter to an impulse 1000

```

y_out <= y;
clk2  <= clk_div2;
x_e <= x_even; -- Monitor some extra test signals
x_o <= x_odd;
y_e <= y_even;
y_o <= y_odd;

Arithmetic: PROCESS(reset, clk_div2)
BEGIN
  IF reset = '1' THEN -- Asynchronous clear
    xd_even <= 0; sum_x_even <= 0; y_even<= 0;
    xd_odd<= 0; sum_x_odd <= 0; y_odd <= 0;
  ELSIF falling_edge(clk_div2) THEN
    xd_even <= x_even;
    sum_x_even <= (xd_even * 2 + xd_even) /4 + x_odd;
    y_even <= (y_even * 8 + y_even )/16 + sum_x_even;
    xd_odd <= x_odd;
    sum_x_odd <= (xd_odd * 2 + xd_odd) /4 + xd_even;
    y_odd  <= (y_odd * 8 + y_odd) / 16 + sum_x_odd;
  END IF;
END PROCESS Arithmetic;

END fpga;

```

The design is realized with two PROCESS statements. In the first, PROCESS Multiplex, x is split into even and odd indexed parts, and the output y is recombined at the clk rate. In addition, the first PROCESS statement generates the second clock, running at $clk/2$. The second block implements the filter's arithmetic according to (4.22). The design uses 236 LEs, no embedded multiplier, and has $F_{max}=479.39$ MHz registered performance using the TimeQuest slow 85C model. The simulation is shown in Fig. 4.20. 4.6

The disadvantage of the parallel implementation, compared with the other methods presented, is the relatively high implementation cost of 236 LEs.

4.4.5 IIR Design Using RNS

Because the residue number system (RNS) uses an intrinsically short word-length, it is an excellent candidate to implement fast (recursive) IIR filters. In a typical IIR-RNS design, a system is implemented as a collection of recursive and nonrecursive systems, each defined in terms of an FIR structure (see Fig. 4.21). Each FIR may be implemented in RNS-DA, using a quarter-square multiplier, or in the index domain, as developed in Chap. 2 (p. 71).

For a stable filter, the recursive part should be scaled to control dynamic range growth. The scaling operation may be implemented with mixed radix conversion, Chinese remainder theorem (CRT), or the ϵ -CRT method. For high-speed designs it is preferable to add an additional pipeline delay based on the clustered or scattered look-ahead pipelining technique [44, Sect. 4-2]. An RNS recursive filter design will be developed in detail in Sect. 5.3. It will be seen that RNS design will improve speed by more than 40%.

4.5 Narrow Band IIR Filter

In the literature many studies can be found that have been done on the required (fractional) bit width for the different IIR architectures. In Sect. 4.3.1 we discussed the Crochiere and Oppenheim [102] results that have been revisited by Dempster and Macleod in the light of the reduced adder graph techniques [103, 104]. A serial or parallel configuration of second order section came out as the winner. Third and fourth place went to the Wave digital filters (WDFs) and Lattice filters.

The underlying filter design by Crochiere and Oppenheim has been a bandpass filter. However, in many IIR designs, in practice lowpass filters are favored. Another important point that is not usually discussed because filters are assumed to have a (unit) gain of one is the overall *integer* bit requirement of the different architectures.

In addition, for narrow band IIR filters a parallel architecture of all-pass filters has often been favored in more recent studies due to the promise of

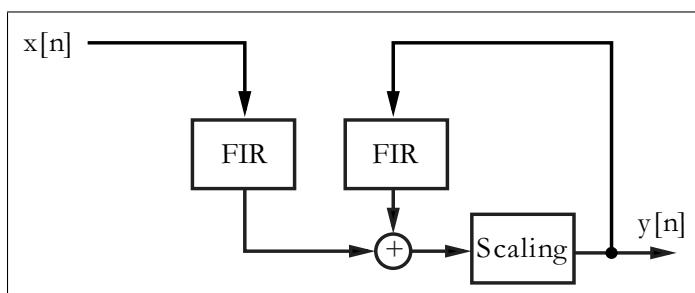


Fig. 4.21. RNS implementation of IIR filters using two FIR sections and scaling

small gains in the amplitudes. Originally developed for WDFs this principle can also be used for second order type all-pass filters or in combination with lattice filters. The most promising architecture are then:

- Transposed direct form I approach
- Direct form second order sections cascade or parallel approach
- Lattice one or two multiplier per section [112]
- Wave digital filter ladder filter [101]
- All-pass transposed direct form I approach
- All-pass direct form second order sections cascade or parallel approach
- All-pass lattice one or two multiplier per section
- All-pass wave digital filter, a.k.a. lattice wave digital filters [113]

In order to get a good overview of these different design choices let us start with a typical narrow band lowpass filter design.

4.5.1 Narrow Band Design Example

IIR filters are typically used when an FIR with the same design specification would be too long, requiring excessive hardware resources and introducing long delays. A filter should be designed that has the following specifications: sampling frequency $60 \times 128 = 7680$ Hz; passband 0–40 Hz, stopband 40–3840 Hz; passband ripple, 1 dB, and stopband ripple, 48–50 dB. The 50 dB stopband ripple is slightly higher than needed of a 8-bit system ($6 \times 8 = 48$ dB) to make room for quantization noise. Such a filter has been used to compute harmonic distortion in power systems [114, 115]. The oversampling rate of the filter is high, while the pass- and stopband are moderate at ca. 8-bit precision only.

The elliptic (or Cauer) filter has the lowest order and will be used as the starting point. It comes out as a fifth order filter, but if we zoom into the pole-zero plot (see Fig. 4.22e) we can imagine the high precision needed to match the tolerance scheme of the transfer function. We first define the lowpass Cauer filter parameters and then compute the filter coefficients using the `ellip` function. The following MATLAB code can be used to generate the coefficients:

```
[B A] = ellip(5,1,50,40/7680*2)
```

and the parameter are filter length, passband ripple in dB, stopband ripple in dB, and the cut-off frequency. Sampling frequency is assumed to be normalized to 2. The forward filter coefficients are placed in array A and the feedback coefficients in array B. We will get the following results using MATLAB for a direct form filter implementation:

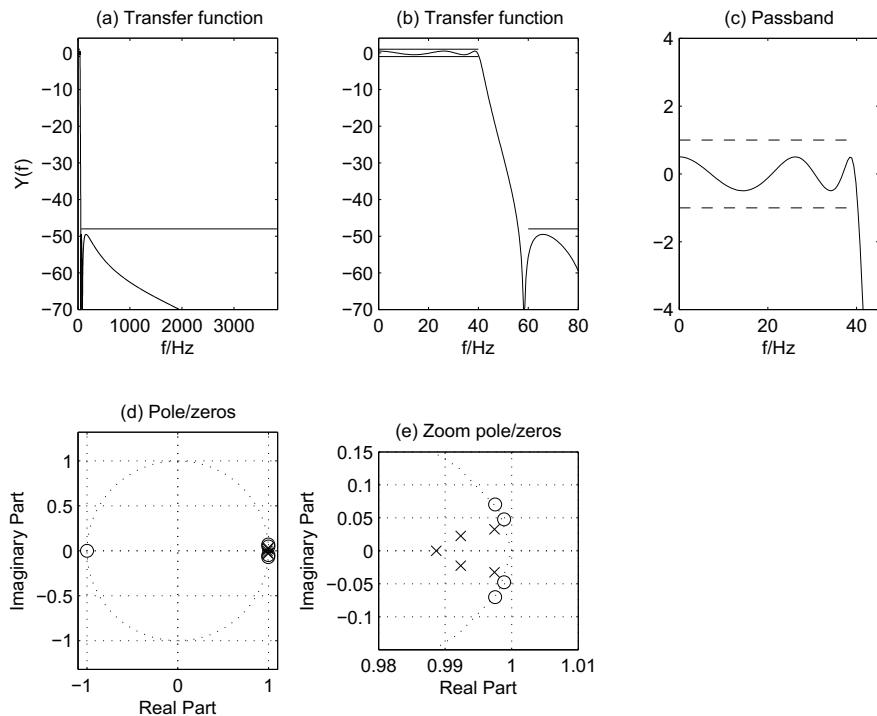


Fig. 4.22. Fifth order elliptic filter showing (a–c) magnitude, (d,e) pole/zero plot

$$B(1) = 0.00030357$$

$$B(2) = -0.00090853$$

$$B(3) = 0.00060496$$

$$B(4) = 0.00060496$$

$$B(5) = -0.00090853$$

$$B(6) = 0.00030357$$

$$A(1) = 1.00000000$$

$$A(2) = -4.96820259$$

$$A(3) = 9.87475368$$

$$A(4) = -9.81500690$$

$$A(5) = 4.87856394$$

$$A(6) = -0.97010812$$

and the overall gain value is $gain = 0.000304$.

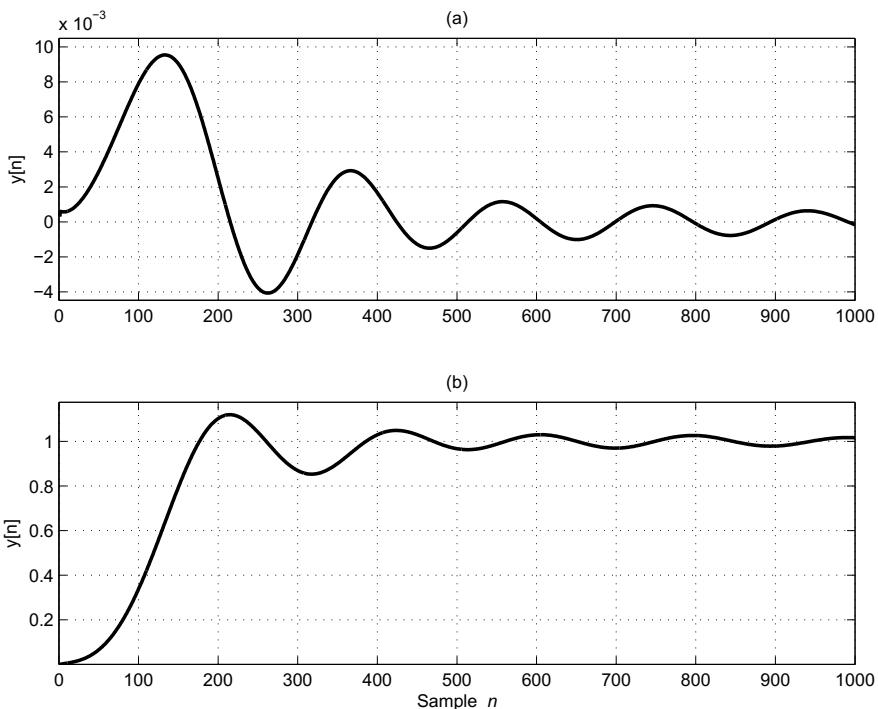


Fig. 4.23. Fifth order elliptic filter showing (a) impulse response, (b) step response

Figure 4.22 shows the transfer function, and the complete pole/zero plot and the “zoom” pole plot of the filter ignoring the zero at $z = -1$. As typical for elliptic filters, all zeros are located on the unit circle.

Since the filter is very narrow the impulse response is quite long. The maximum gain of any filter can be found if we use the eigenfrequencies as filter input signal. For a lowpass filter that is found at DC or $f = 0$ frequency, the input signal is a step function and the output the so-called step response. Impulse and step response are shown in Fig. 4.23.

Using the pole/zero values we then make our architecture choice(s) and can then simulate the filter(s) in SIMULINK to verify the correct behavior. In SIMULINK we can clearly count the required resources and determine the longest path and suggest additional pipeline registers to improve the speed of the filter. This additional pipeline registers may increase the overall delay, but as long as the other pole/zero locations are not changed the transfer function will remain the same.

To implement the filter in fixed-point arithmetic we need to determine the required number of fractional and integer bits for each adder and (coefficient) multiplier. We need to distinguish between a design with embedded multipliers and using LEs. With LEs we can use ternary values (i.e., $-1, 0, 1$) CSD

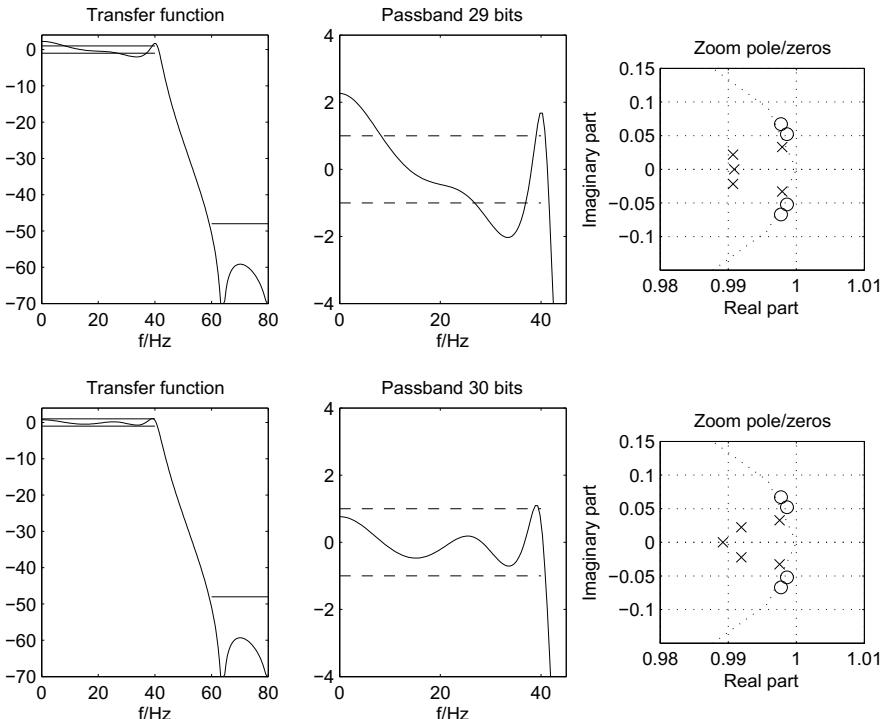


Fig. 4.24. Transfer functions for quantized coefficients of the direct form filter

coding and combine multiple coefficients in a reduced adder graph (RAG-n). In both cases we need first to get an estimate of the required integer and fractional bit width and then fine tune the coefficients with CSD or RAG coding in case of the LE-based implementation.

To estimate the fractional precision of the filter we would need to quantize the coefficient to N bit fractional precision. This can be done by multiplication by 2^N followed by rounding and division by 2^N . We then use this quantized coefficient (A_q, B_q) and determine the transfer function of the filter. If the tolerance scheme is met we are done; otherwise we increase N by one ($N \rightarrow N + 1$) and try again.

Estimation of the required integer precision can be done without iterations. We just determine the “eigenfrequency” of the filter and apply this eigenfrequency to the filter input. For a narrow band lowpass filter a reasonable assumption is that $f = 0$ is the eigenfrequency and we would then use the step function as input signal. Then we take the \log_2 of the maximum amplitude of each adder to determine the required integer bits at each node.

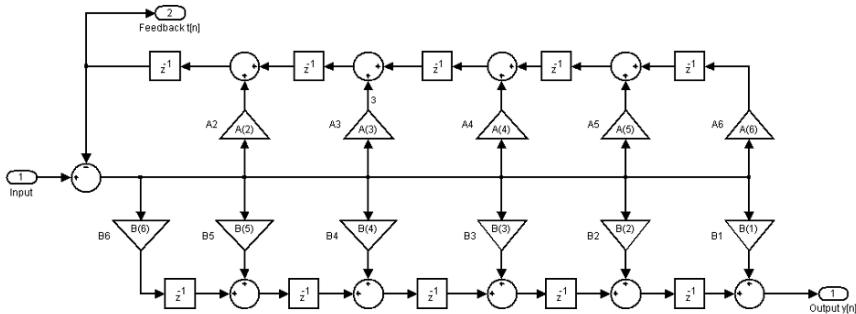


Fig. 4.25. Transposed direct form I design in SIMULINK

Example 4.7: Bit Width Requirement of Narrow IIR Filter

We take the filter coefficient for feedback coefficients A and feedforward coefficients B from above and increase the number of fractional bits until the specified tolerance scheme is met. From the transfer function shown in Fig. 4.24 it can be concluded that 30-bit fractional precision is required.

For the integer bits the filter is designed in SIMULINK (or MATLAB) and then the step-response is measured; see Fig. 4.25. For the measurement of the feedback gain shown in Fig. 4.26 we conclude that $\lceil \log_2(6 \times 10^9) \rceil = \lceil 32.48 \rceil = 33$ -bit internal integer precision is required to avoid overflow in arithmetic. Figure 4.26c shows the \log_2 of the maximum magnitude at each adder output.

Together the data path for the direct filter design including a sign bit would require $1 + 30 + 33 = 64$ bits, which make a design using the VHDL INTEGER data type impossible. The direct form exceeds therefore the possible INTEGER data type bit width (i.e., 32 bits) and a complete STD_LOGIC_VECTOR implementation would be necessary that is usually much more cumbersome than an integer data type based design. The new sfixed VHDL-2008 data types simplify the design as is shown in the VHDL code⁴ example next.

```
-- 
-- Description: This is a 5th order IIR in direct form
-- implementation.
-- Feedforward coefficients B=
--   0.000304 -0.000909 0.000605
--   0.000605 -0.000909 0.000304
-- Feedback coefficients A=
--   1.000000 -4.968203 9.874754
--   -9.815007 4.878564 -0.970108
--
LIBRARY ieee;
USE ieee.std_logic_1164.ALL; USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

LIBRARY ieee_proposed;
USE ieee_proposed.fixed_float_types.ALL;
```

⁴ The equivalent Verilog code `iir5sf.v` for this example can be found in Appendix A on page 824. Synthesis results are shown in Appendix B on page 881.

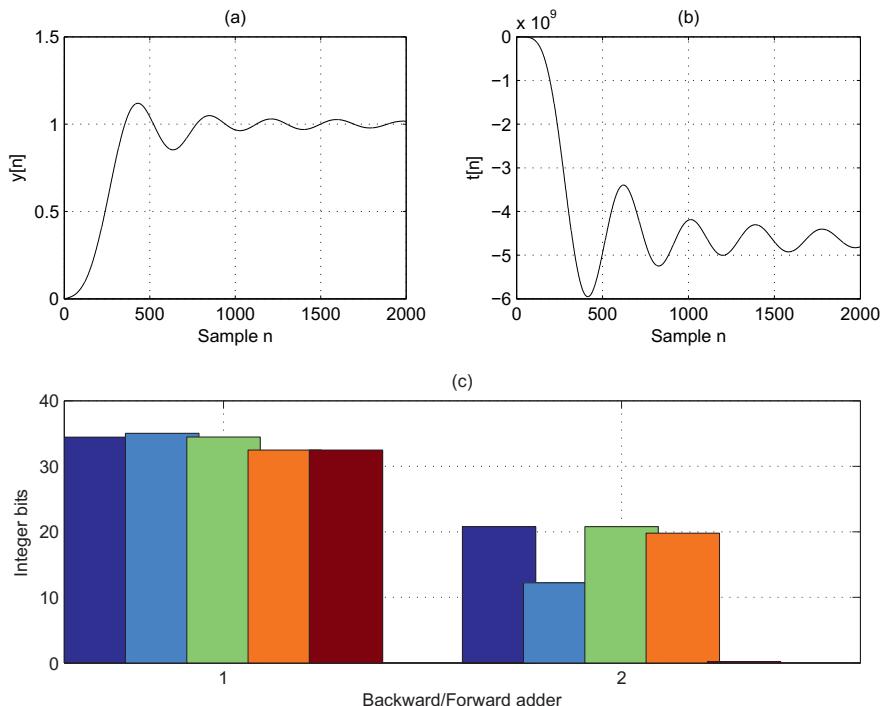


Fig. 4.26. (a) Filter output $y[n]$. (b) The feedback signal $t[n]$. (c) Binary logarithm of all 10 adder amplitude signals in the architecture

```

USE ieee_proposed.fixed_pkg.ALL;
USE ieee_proposed.float_pkg.ALL;
-----
ENTITY iir5sfix IS           -----> Interface
  PORT (clk      : IN STD_LOGIC; -- System clock
        reset    : IN STD_LOGIC; -- System reset
        switch   : IN STD_LOGIC; -- Feedback switch
        x_in     : IN STD_LOGIC_VECTOR(63 DOWNTO 0);          -- System input
        t_out    : BUFFER STD_LOGIC_VECTOR(39 DOWNTO 0);        -- Feedback
        y_out    : OUT STD_LOGIC_VECTOR(39 DOWNTO 0));         --System output
  END;
-----
ARCHITECTURE fpga OF iir5sfix IS

  CONSTANT a2 : SFIXED(4 DOWNTO -30) :=
                TO_SFIXED(-4.9682025852, 4,-30);
  CONSTANT a3 : SFIXED(5 DOWNTO -30) :=
                TO_SFIXED(9.8747536754, 5,-30);
  CONSTANT a4 : SFIXED(5 DOWNTO -30) :=
                TO_SFIXED(-9.8150069021, 5,-30);

```

```

CONSTANT a5 : SFIXED(4 DOWNTO -30) :=
          TO_SFIXED(4.8785639415, 4,-30);
CONSTANT a6 : SFIXED(1 DOWNTO -30) :=
          TO_SFIXED(-0.9701081227, 1,-30);
CONSTANT b1 : SFIXED(0 DOWNTO -30) :=
          TO_SFIXED(0.0003035737, 0,-30);
CONSTANT b2 : SFIXED(0 DOWNTO -30) :=
          TO_SFIXED(-0.0009085259, 0,-30);
CONSTANT b3 : SFIXED(0 DOWNTO -30) :=
          TO_SFIXED(0.0006049556, 0,-30);
CONSTANT b4 : SFIXED(0 DOWNTO -30) :=
          TO_SFIXED(0.0006049556, 0,-30);
CONSTANT b5 : SFIXED(0 DOWNTO -30) :=
          TO_SFIXED(-0.0009085259, 0,-30);
CONSTANT b6 : SFIXED(0 DOWNTO -30) :=
          TO_SFIXED(0.0003035737, 0,-30);
SIGNAL h, s1, s2, s3, s4, s5 :
          SFIXED(33 DOWNTO -30) := (OTHERS => '0');
SIGNAL x, y, t, r2, r3, r4, r5 :
          SFIXED(33 DOWNTO -30) := (OTHERS => '0');
SIGNAL y_sfix, t_sfix : SFIXED(23 DOWNTO -16);

BEGIN

  x <= TO_SFIXED(x_in, x); -- Redefine bits as signed
                           -- FIX 34.30 number
  P1: PROCESS (reset, clk, x, t, s1, s2, s3, s4, s5, r2,
               r3, h, switch)
BEGIN    -- First equations without inferring registers
  IF switch = '0' THEN
    h <= x; -- Switch is open
  ELSE
    h <= resize(x - t, x, fixed_wrap,fixed_truncate);
  END IF;                                -- Switch is closed
  IF reset = '1' THEN -- Reset all register
    t <= (OTHERS => '0'); y <= (OTHERS => '0');
    r2 <= (OTHERS => '0'); r3 <= (OTHERS => '0');
    r4 <= (OTHERS => '0'); r5 <= (OTHERS => '0');
    s1 <= (OTHERS => '0'); s2 <= (OTHERS => '0');
    s3 <= (OTHERS => '0'); s4 <= (OTHERS => '0');
    s5 <= (OTHERS => '0');
  ELSIF rising_edge(clk) THEN -- IIR in direct form
    -- Using the "do not WRAP"
    r5 <= resize(      a6 * h,x,fixed_wrap,fixed_truncate);
    r4 <= resize(r5 + a5 * h,x,fixed_wrap,fixed_truncate);
    r3 <= resize(r4 + a4 * h,x,fixed_wrap,fixed_truncate);
    r2 <= resize(r3 + a3 * h,x,fixed_wrap,fixed_truncate);
    t  <= resize(r2 + a2 * h,x,fixed_wrap,fixed_truncate);
    s5 <= resize(      b6 * h,x,fixed_wrap,fixed_truncate);
    s4 <= resize(s5 + b5 * h,x,fixed_wrap,fixed_truncate);
    s3 <= resize(s4 + b4 * h,x,fixed_wrap,fixed_truncate);
    s2 <= resize(s3 + b3 * h,x,fixed_wrap,fixed_truncate);
    s1 <= resize(s2 + b2 * h,x,fixed_wrap,fixed_truncate);
  END IF;
END PROCESS P1;

```

```

      y  <= resize(s1 + b1 * h,x,fixed_wrap,fixed_truncate);
    END IF;
END PROCESS;

-- Convert to 24.16 sfixed number
y_sfix  <= resize(y, y_sfix,fixed_wrap,fixed_truncate);
t_sfix  <= resize(t, t_sfix,fixed_wrap,fixed_truncate);
-- Redefine bits as 40 bit SLV
y_out <= to_slv(y_sfix);
t_out <= to_slv(t_sfix);

END fpga;

```

From the VHDL code the simplification due to the **sfixed** library can be seen. The coefficients and the input **x** are translated into HDL with the **TO_SFIXED** function. Then the **PROCESS** for the fifth order filter follows. We first include an asynchronous reset to all registers and a multiplexer so that the feedback path can be turned off. The feedback switch is useful in the debugging phase since with an open switch the feedback signals **t_out** of the impulse response will show the feedback coefficients one at a time. After the **rising_edge** statement the 11 assignments for the filter follow. We apply the **resize** function after each arithmetic operation to avoid overflow or bit growth. The parameter **fixed_wrap** and **fixed_truncate** are used to avoid the default thresholding in the **resize** function. Finally we convert the feedback signal **t** and the filter output **y** to **sfixed** and then to SLV output format.

The design uses 2474LEs, 128 embedded multipliers, and has an **Fmax** = 46.99 MHz registered performance using the **TimeQuest** slow 85C model. The response of the filter to an impulse of amplitude $1.0 = 40000000_{16}$ for 34.30 **sfixed** format, shown in Fig. 4.27, agrees with the MATLAB simulated results presented in Fig. 4.23a (p. 253). The 60 μ s simulation requires 600 clock cycles for a 100-ns system clock. The impulse maximum occurs after about 133 clock cycles.

4.7

If we had not used the parameter **fixed_wrap** and **fixed_truncate** to avoid the default thresholding then the design would be larger (6279 LEs) and slower (29.96 MHz). If we had used the 64-bit floating type from VHDL-2008 the design would have a little easier coding since no **resize** function call would be necessary. However the design would be substantially larger (over 42,682 LEs and 180 multipliers) and run much slower (6.5 MHz). We hope that with serial or parallel BiQuad sections substantial reductions in bit width is possible that will allow an **INTEGER** data type-based implementation in VHDL.

When implementing the filter with LEs additional filter optimization steps are usually added. Several method have been suggested in the literature and we would like to mention briefly the most popular. Some methods can also be combined.

- 1) With LEs implementation the binary coding of the multiplier is replaced by signed-power-of-two factors. A reasonable approach is then to start

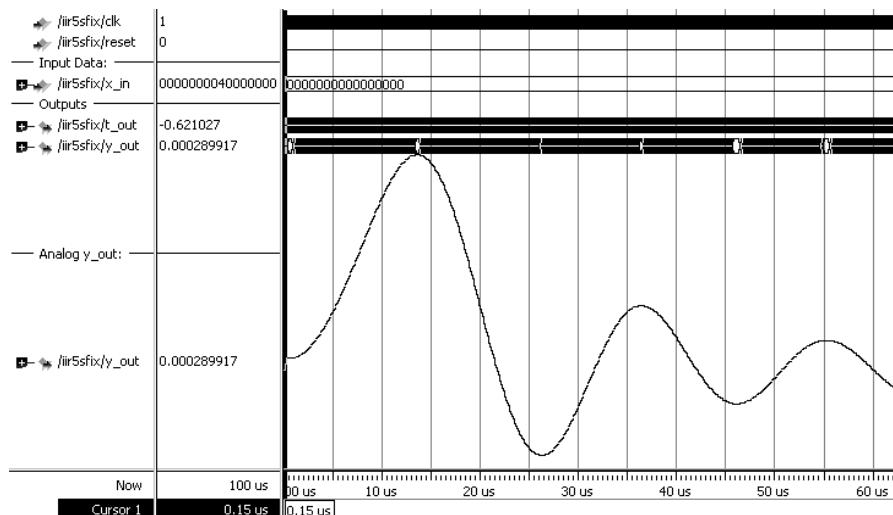


Fig. 4.27. Impulse response for ModelSim simulation of the fifth order Cauer filter in direct form

with a few (e.g., two) signed-power-of-two factors for each coefficient and then add more successively until the filter tolerance scheme is met.

- 2) Another approach is to use the integer values for each coefficient determined earlier and then iteratively change the coefficient by a small amount (± 1) and check whether that change has a benefit to the transfer function. This can be done by nonlinear learning algorithms such as the Integer Linear Programming (ILP) or a Genetic Algorithm; see `fminimax` from the MATLAB optimization toolbox.
- 3) If several filter coefficients have the same input then and only then can the Reduced Adder Graph method be beneficial. However, since in the feedback part of the IIR filter no additional registers can be introduced, it is best to use RAG with minimum adder depth. Also the scaling factor is usually fixed in the RAG algorithm such that fine adjustment of the coefficients can no longer be done.

Some of the methods can be combined, such as 2 and 3, but bear in mind that such nonlinear optimization cannot necessarily find the global minimum of the solution and that savings are moderate when compared to architecture choices.

4.5.2 Cascade Second Order Systems Narrow Band Filter Design

The implementation of narrow band IIR filters using cascades of first and second order systems has several benefits. The obvious is that with design of pole/zero pairs in a BiQuad the gain can be chosen such that the quantization

noise is reduced and at the same time the integer gain does not become too large. Since elliptic filters provide the shortest filter length and are often preferred an additional benefit arises from the fact that zeros are on the unit circle and therefore the second order zero sections are of the type $1 + b(k, 1)z^{-1} + z^{-2}$, reducing the number of multipliers.

When implementing the second order sections an immediate question arises: which pole/zero should we combine in one section and how should we order the sections. Jackson [116] has compiled some rules of thumb that we should follow. Since the poles closest to the unit circle have the largest gains and produce the greatest ripples it follows that we should start with the closest pole to the unit circle first to combine this with the closest zero to reduce the gain and the possibility of overflow in the arithmetic. Then we proceed with the second closest poles etc. Now since the closest pole give the most ripple, we place this pole section *last* in the cascade, or put differently, the pole furthest away from the unit circle is implemented first. If we look at the pole/zero plot in Fig. 4.22 we see that the first order section pole (i.e., that on the real axis) has the most distance to the unit circle, so this section comes first. Then come the two second order systems. The MATLAB function `tf2sos` already provides the second order sections in this recommended order. We run the MATLAB function as follows:

```
[sos gain] = tf2sos(B,A);
```

i.e., using the forward filter coefficient from array B and the feedback coefficients from array A we will get the following results using MATLAB for the second order sections from `tf2sos`:

$b[i, 1]$	$b[i, 2]$	$b[i, 3]$	$a[i, 1]$	$a[i, 2]$	$a[i, 3]$
1.0000	1.0000	0	1.0000	-0.9887	0
1.0000	-1.9950	1.0000	1.0000	-1.9847	0.9852
1.0000	-1.9977	1.0000	1.0000	-1.9948	0.9959

Note that with `tf2sos` only one overall gain factor is given as 0.000304. To avoid excessive quantization noise the overall gain needs to be distributed over the three stages. We may scale each output in such a way that the filter amplitude $|H_k(\omega)| \leq 1.0$. A more practical approach for the lowpass filter is to normalize the amplitude gain to one for the step response at each filter output. Then the individual scaling factors become $s(1) = 0.0057$, $s(2) = 0.1147$, and $s(3) = 0.4677$, with $s(1)s(2)s(3) = \text{gain} = 0.000304$.

We can now implement the BiQuad sections in the direct form, such that the maximum RAG block can be built as shown in Fig. 4.28.

The required fractional bits are computed by the same procedure we used for the direct form filter, i.e., each coefficient is quantized and we measure whether the tolerance scheme of the transfer function is matched. As can be seen from Fig. 4.29, only 14 bits are required to match the tolerance scheme. This compares favorably to the 30 bits required by the direct form filter.

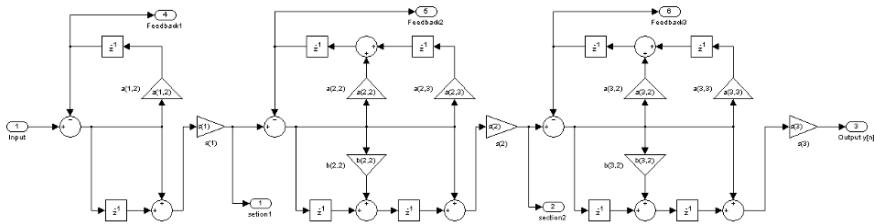


Fig. 4.28. Fifth order elliptic filter implemented with direct form II BiQuad sections and individual scaling

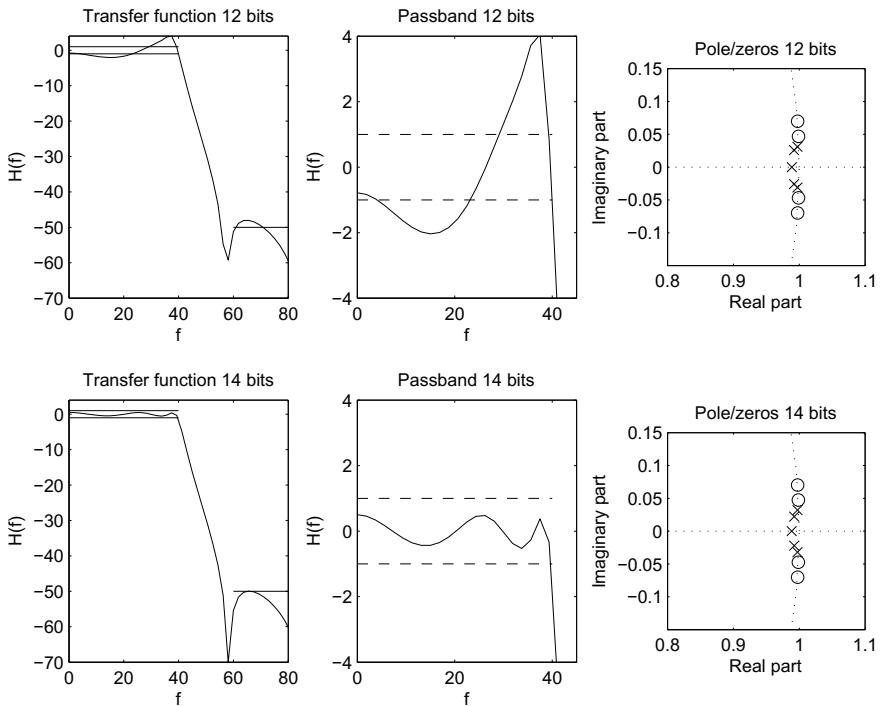


Fig. 4.29. Transfer functions and pole zero plots for quantized coefficients of the second order section design

To determine the required number of integer bits of the filter we use the same procedure as for the direct form filter, i.e., the filter is designed in SIMULINK and then the step-response is measured; see Fig. 4.30a. The binary logarithm of the maximum amplitude for each adder gives the required integer bits; see Fig. 4.30c. For the measurement of the feedback gain shown in Fig. 4.30b we conclude that worst case $\lceil \log_2(1,700) \rceil = \lceil 10.7 \rceil = 11$ bit internal integer precision is required to avoid overflow in arithmetic. This

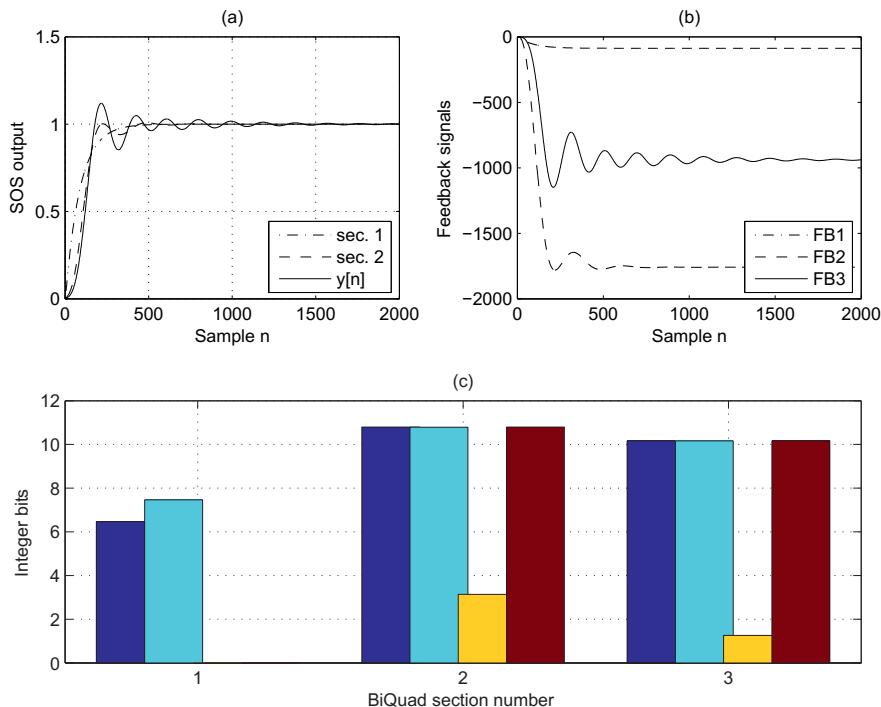


Fig. 4.30. Integer bit width computation for the BiQuad cascade second order design

compares favorably with the 33 bits required in the direct form implementation.

A small improvement in the required integer bit width is possible if we (a) give up the idea that we can optimize *all* BiQuad coefficients with a single RAG algorithm and (b) allow some extra delay. The key idea of the modified BiQuad section is that we implement the zeros first in order to reduce the amplitude for the feedback part of the system. This is shown for one second order system in Fig. 4.31. The “zeros” will be implemented first and then the pole section next. To compute the transfer function of the modified BiQuad let us call the output register of the feedforward part $w[n]$; then we get in the z domain

$$W(z) = X(z)z^{-1} (b[1] + b[2]z^{-1} + b[3]z^{-2}). \quad (4.23)$$

Using the same $W(z)$ it follows then for the output

$$\begin{aligned} Y(z) &= W(z)z^{-2} - Y(z)a[2]z^{-1} - Y(z)a[3]z^{-2} \\ &= \frac{W(z)z^{-2}}{1 + a[2]z^{-1} + a[3]z^{-2}} \end{aligned} \quad (4.24)$$

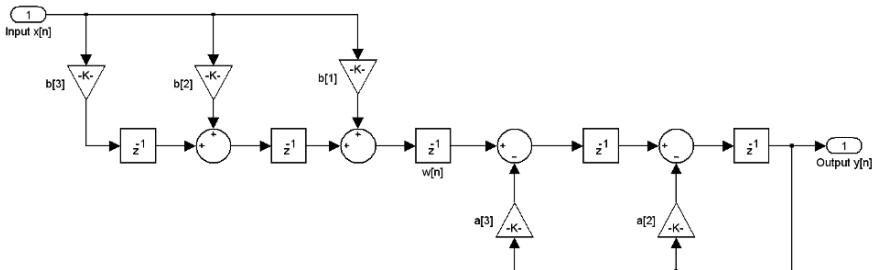


Fig. 4.31. Modified BiQuad system to reduce the integer gain

and the transfer function of the modified BiQuad becomes

$$F(z) = \frac{Y(z)}{X(z)} = \frac{b[1] + b[2]z^{-1} + b[3]z^{-2}}{1 + a[2]z^{-1} + a[3]z^{-2}} z^{-3}, \quad (4.25)$$

i.e., it is the usual BiQuad transfer function (see Fig. 4.11, p. 236) with an extra delay by three samples.

One additional benefit of the modified BiQuad is that the worst case delay is also reduced by one adder to one multiplier and one adder. If we now measure again the required integer bit width we see from Fig. 4.32 the reduced length in the integer bit width.

4.5.3 Parallel Second Order Systems Narrow Band Filter Design

When compared to cascade implementation, parallel implementation using second order systems has advantages and disadvantages. On the plus side we do not need to be concerned about the ordering of the sections since we have a parallel architecture. One disadvantage usually associated with the design of parallel elliptic filters is that now less coefficients are trivial (± 1) due to the partial-fraction expansion of the system function as the following demonstration will show. To accomplish a partial-fraction expansion in MATLAB the `residue` function is used, i.e.,

```
[R, P, D] = residue(B,A)
```

where the vector R contains the residue, vector P the poles, and scalar D the direct term, i.e.,

$$F(z) = D + \sum_{k=1}^N \frac{R(k)}{z - P(k)} \quad (4.26)$$

For our elliptic fifth order filter we will get the following values:

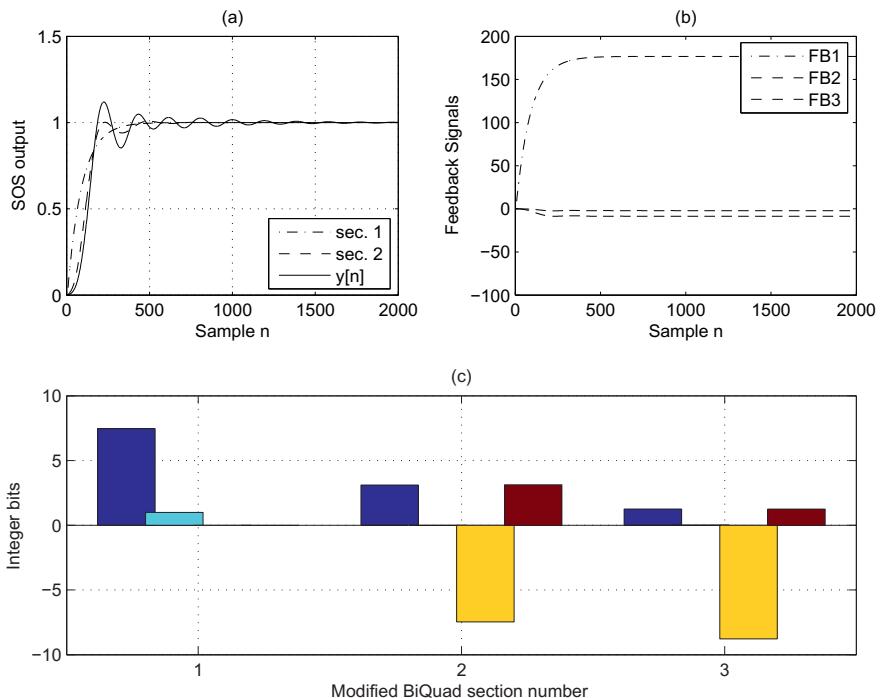


Fig. 4.32. Integer bit width computation for the modified BiQuad second order design

$$R(1, 2) = 0.0015 \pm j0.0007$$

$$R(3, 4) = -0.0073 \pm j0.0026$$

$$R(5) = 0.0122$$

$$P(1, 2) = 0.9974 \pm 0.0325$$

$$P(3, 4) = 0.9923 \pm 0.0226$$

$$P(5) = 0.9887$$

and as scalar $D = 0.00030357$. Since we like to avoid building a system with complex poles we combine the two conjugate complex poles. For the first pair we get

$$\frac{R(1)}{z - P(1)} + \frac{R(2)}{z - P(2)} = \frac{R(1)(z - P(2)) + R(2)(z - P(1))}{(z - P(1))(z - P(2))} \quad (4.27)$$

$$= \frac{(R(1) + R(2))z + R(1)P(2) + R(2)P(1)}{z^2 + 2z\Re(P(1)) + |P(1)|^2} \quad (4.28)$$

In MATLAB combining transfer system functions can be reconstructed using the `residue` function now with three input parameters as follows:

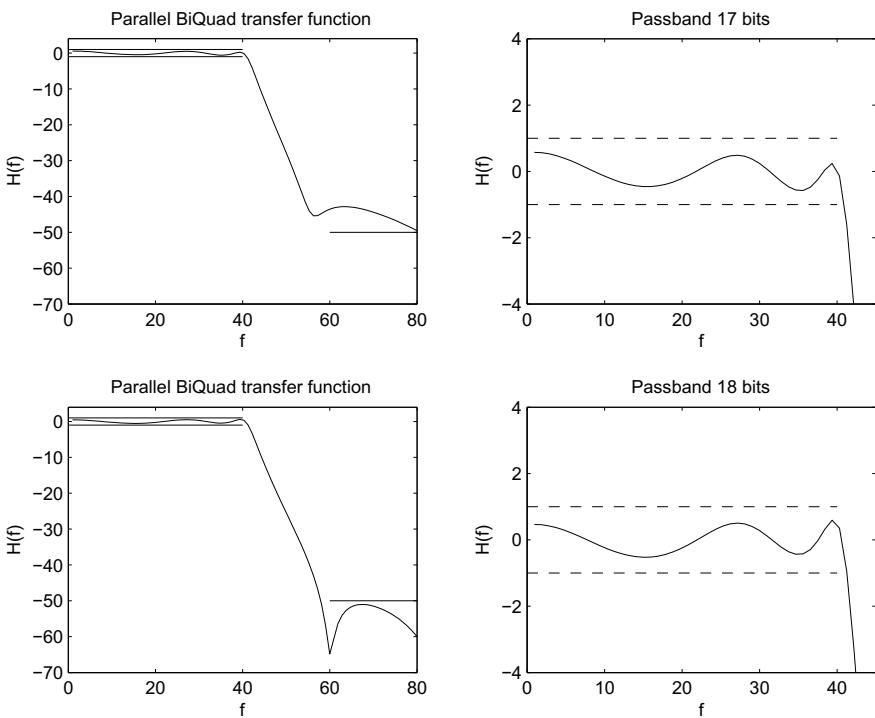


Fig. 4.33. Transfer functions plots for quantized coefficients of the parallel second order section design

```
[B1,A1] = residue([R(1) R(2)], [P(1) P(2)], 0)
```

and we get the following values:

$$\begin{aligned} B1 &= 0.0031 & -0.0032 & 0 \\ A1 &= 1.0000 & -1.9948 & 0.9959 \end{aligned}$$

and for the second biquad section we use

```
[B2,A2] = residue([R(3) R(4)], [P(3) P(4)], 0)
```

and we get the following values:

$$\begin{aligned} B2 &= -0.0146 & 0.0146 & 0 \\ A2 &= 1.0000 & -1.9847 & 0.9852 \end{aligned}$$

Note that when combining the two conjugate complex pole sections to one real system the resulting numerator becomes a multiplication of a constant factor with a first order polynomial. As a consequence the numerator is of first order and not second order as the denominator and we save one multiplier and one adder in the implementation of the zeros. We can now compose the SIMULINK model of the filter to compute the integer bit width. The computation of the

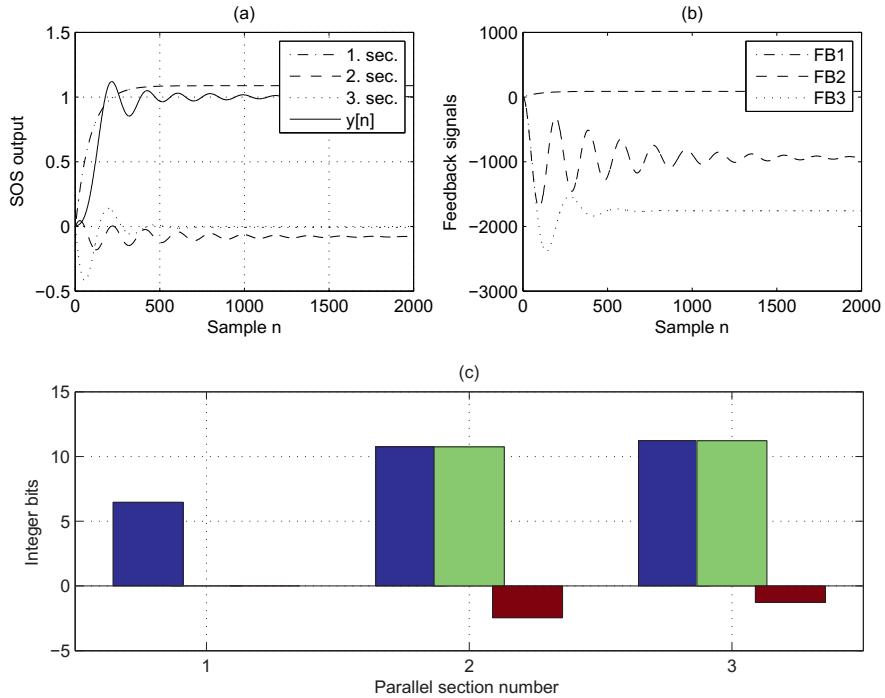


Fig. 4.34. Integer bit width computation for the parallel BiQuad second order system design

fractional bits and the required integer bits works as for the direct form filter and the results are shown in Figs. 4.33 and 4.34, respectively. From Fig. 4.33 it can be seen that for 17-bit quantization the stop band is not matching the tolerance scheme, but for 18-bit precision both pass and stopband are a match. The standard parallel BiQuad implementation shows a maximum integer gain of about 11 bits; see Fig. 4.34.

Again a few bits can be saved if we use the modified BiQuad section from Fig. 4.31. The resulting integer bit width is shown in Fig. 4.35 and shows that no integer bits are required at all for the implementation with the modified BiQuad sections. However, note that the modified BiQuad of different orders show different delays. We need therefore to add a delay of three samples for the direct part, and a delay of one for the first order section since the BiQuad has a delay of three samples as (4.25) shows. That can be verified by monitoring the impulse responses of all four subsystems and comparing with the standard BiQuad section. Figure 4.36 shows the overall system for the fifth order modified parallel BiQuad system.

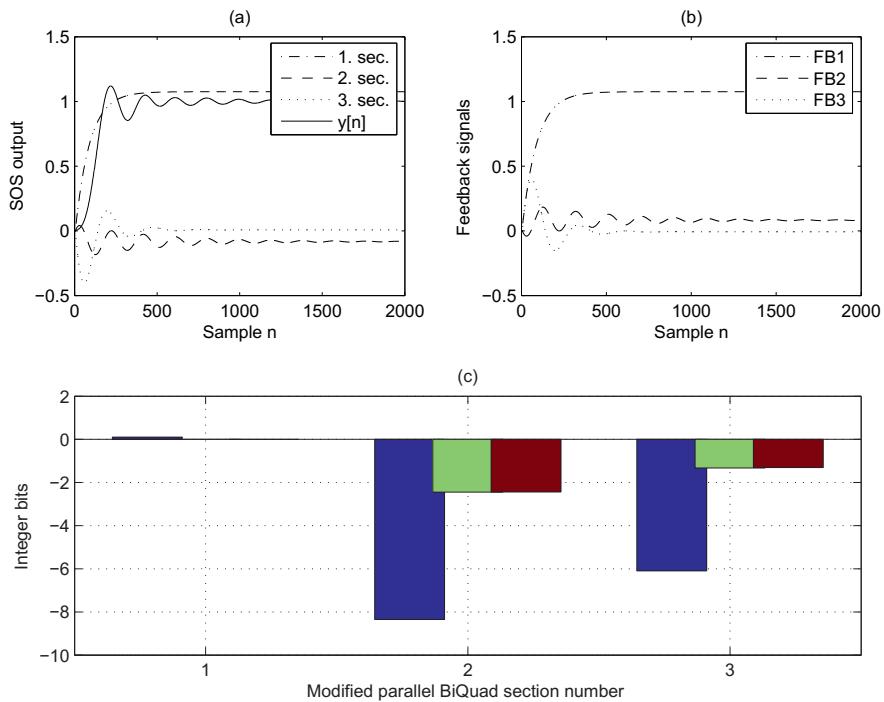


Fig. 4.35. Integer bit width computation for the parallel modified BiQuad second order design

Since this parallel design using the modified BiQuad has several attractive features and short word length let us have a look at an HDL implementation next.

Example 4.8: Modified Parallel BiQuad IIR Filter

We use the filter coefficients we just computed with the help of the `residue` function. Since we like to test impulse and step response we will use a 2.19 `sfixed` internal format, i.e., two integer bits and 19 fractional bits. Each decimal digit in our constants will increase the precision by 3.32 bits and for 19 bits we therefore need at least $\lceil 19/3.32 \rceil = 6$ decimal digits for each coefficient. In MATLAB we can increase the digits displayed using the `sprintf` function since by default only four digits are displayed.

The following VHDL code⁵ shows a possible implementation of this fifth order IIR filter using parallel BiQuad sections:

```
-- Description: 5th order IIR parallel form implementation
-- Coefficients:
-- D = 0.00030357
-- B1 = 0.0031 -0.0032 0
```

⁵ The equivalent Verilog code `iir5para.v` for this example can be found in Appendix A on page 825. Synthesis results are shown in Appendix B on page 881.

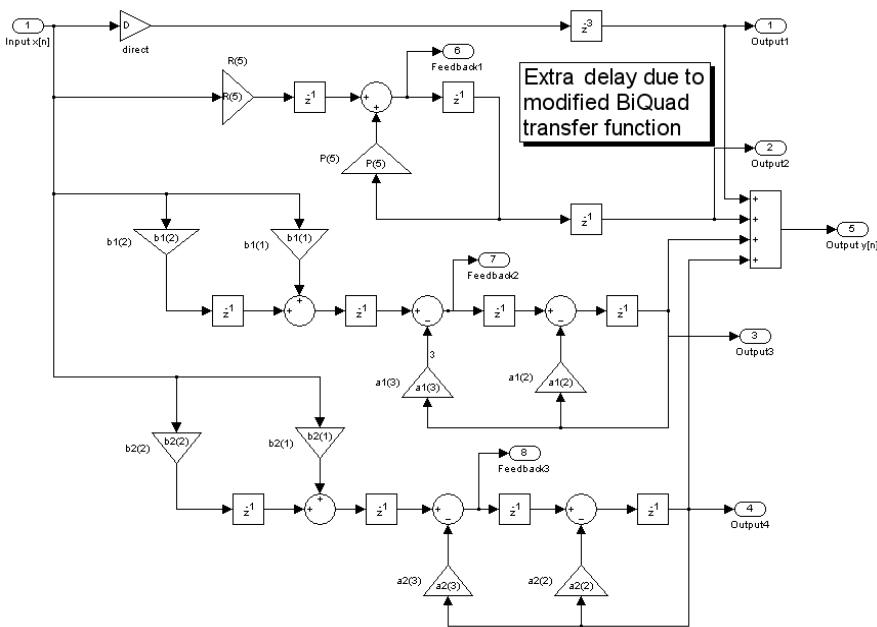


Fig. 4.36. Modified parallel BiQuad section design in SIMULINK

```
-- A1 = 1.0000 -1.9948 0.9959
-- B2 = -0.0146 0.0146 0
-- A2 = 1.0000 -1.9847 0.9852
-- B3 = 0.0122
-- A3 = 0.9887
```

```

ARCHITECTURE fpga OF iir5para IS
  -- SUBTYPE FIX : IS SFIXED(1 DOWNTO -18);
-- First BiQuad coefficients
  CONSTANT a12 : SFIXED(1 DOWNTO -18)
    := TO_SFIXED(-1.99484680, 1,-18);
  CONSTANT a13 : SFIXED(1 DOWNTO -18)
    := TO_SFIXED(0.99591112, 1,-18);
  CONSTANT b11 : SFIXED(1 DOWNTO -18)
    := TO_SFIXED(0.00307256, 1,-18);
  CONSTANT b12 : SFIXED(1 DOWNTO -18)
    := TO_SFIXED(-0.00316061, 1,-18);
-- Second BiQuad coefficients
  CONSTANT a22 : SFIXED(1 DOWNTO -18)
    := TO_SFIXED(-1.98467605, 1,-18);
  CONSTANT a23 : SFIXED(1 DOWNTO -18)
    := TO_SFIXED(0.98524428, 1,-18);
  CONSTANT b21 : SFIXED(1 DOWNTO -18)
    := TO_SFIXED(-0.01464265, 1,-18);
  CONSTANT b22 : SFIXED(1 DOWNTO -18)
    := TO_SFIXED(0.01464684, 1,-18);
-- First order system with R(5) and P(5)
  CONSTANT a32 : SFIXED(1 DOWNTO -18)
    := TO_SFIXED(0.98867974, 1,-18);
  CONSTANT b31 : SFIXED(1 DOWNTO -18)
    := TO_SFIXED(0.012170, 1,-18);
-- Direct system
  CONSTANT D : SFIXED(1 DOWNTO -18)
    := TO_SFIXED(0.000304, 1,-18);
-- Internal signals
  SIGNAL s11, s12, s21, s22, s31 :
    SFIXED(1 DOWNTO -18) := (OTHERS => '0');
  SIGNAL x, y, r12, r13, r22, r23, r32 :
    SFIXED(1 DOWNTO -18) := (OTHERS => '0');
  SIGNAL r41, r42, r43 :
    SFIXED(1 DOWNTO -18) := (OTHERS => '0');
  SIGNAL x32, y_sfix, y_D, y_1, y_21, y_22
    : SFIXED(15 DOWNTO -16);

BEGIN

  x32 <= TO_SFIXED(x_in, x32); -- Redefine as 16.16 format
  x <= resize(x32, x); -- Internal precision is 2.19 format

  P1: PROCESS (clk, x, reset)      -----> Behavioral Style
BEGIN    -- First equations without inferring registers
  IF reset = '1' THEN -- Reset all register
    y <= (OTHERS => '0');
    r12 <= (OTHERS => '0'); r13 <= (OTHERS => '0');
    r22 <= (OTHERS => '0'); r23 <= (OTHERS => '0');
    r32 <= (OTHERS => '0');
    r41 <= (OTHERS => '0'); r42 <= (OTHERS => '0');
    r43 <= (OTHERS => '0');

```

```

        s11 <= (OTHERS => '0'); s12 <= (OTHERS => '0');
        s21 <= (OTHERS => '0'); s22 <= (OTHERS => '0');
        s31 <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN -- SOS modified BiQuad form
-- 1. BiQuad is 2. order
    s12 <= resize( b12 * x,x,fixed_wrap,fixed_truncate);
    s11 <= resize(s12+b11*x,x,fixed_wrap,fixed_truncate);
    r13 <= resize(s11-a13*r12,x,fixed_wrap,fixed_truncate);
    r12 <= resize(r13-a12*r12,x,fixed_wrap,fixed_truncate);
-- 2. BiQuad is 2. order
    s22 <= resize( b22 * x,x,fixed_wrap,fixed_truncate);
    s21 <= resize(s22+b21*x,x,fixed_wrap,fixed_truncate);
    r23 <= resize(s21-a23*r22,x,fixed_wrap,fixed_truncate);
    r22 <= resize(r23-a22*r22,x,fixed_wrap,fixed_truncate);
-- 3. Section is 1. order
    s31 <= resize( b31 * x,x,fixed_wrap,fixed_truncate);
    r32 <= resize(s31+a32*r32,x,fixed_wrap,fixed_truncate);
-- 4. Section is constant
    r41 <= resize( D * x,x,fixed_wrap,fixed_truncate);
-- Output adder tree
    r42 <= r41;
    r43 <= resize(r42 + r32,x,fixed_wrap,fixed_truncate);
    y <= resize(r12+r22+r43,x,fixed_wrap,fixed_truncate);
    END IF;                                     -- Output sum
END PROCESS;

-- Convert to 16.16 sfixed number
y_sfix <= resize(y, y_sfix,fixed_wrap,fixed_truncate);
y_D <= resize(r42, y_sfix,fixed_wrap,fixed_truncate);
y_1 <= resize(r32, y_sfix,fixed_wrap,fixed_truncate);
y_21 <= resize(r22, y_sfix,fixed_wrap,fixed_truncate);
y_22 <= resize(r12, y_sfix,fixed_wrap,fixed_truncate);
-- Redefine bits as 32 bit SLV
y_out <= to_slv(y_sfix);
y_Dout <= to_slv(y_D);
y_1out <= to_slv(y_1);
y_21out <= to_slv(y_21);
y_22out <= to_slv(y_22);

END fpga;

```

The design ENTITY includes, besides the system input and outputs, the output of the four parallel sections: one direct, one first order, and two second order sections are designed. The CONSTANT coefficients are conveniently defined using the TO_SFIXED function. A single PROCESS for all four systems follows. We first code an asynchronous reset to all registers. After the `rising_edge` statement the assignments for the filter follow. Each assignment will infer a register. We apply the `resize` function after each arithmetic operation to avoid bit growth. The parameter `fixed_wrap` and `fixed_truncate` are used to avoid the default thresholding. Finally we convert the four section signals and filter output to `sfixed` and then to SLV data type.

The design uses 624 LEs, 51 embedded multipliers, and has $F_{max} = 87.69$ MHz registered performance using the TimeQuest slow 85C model. The step response of the filter with amplitude $1.0 = 65536_{10}$ for 16.16 `sfixed` format,

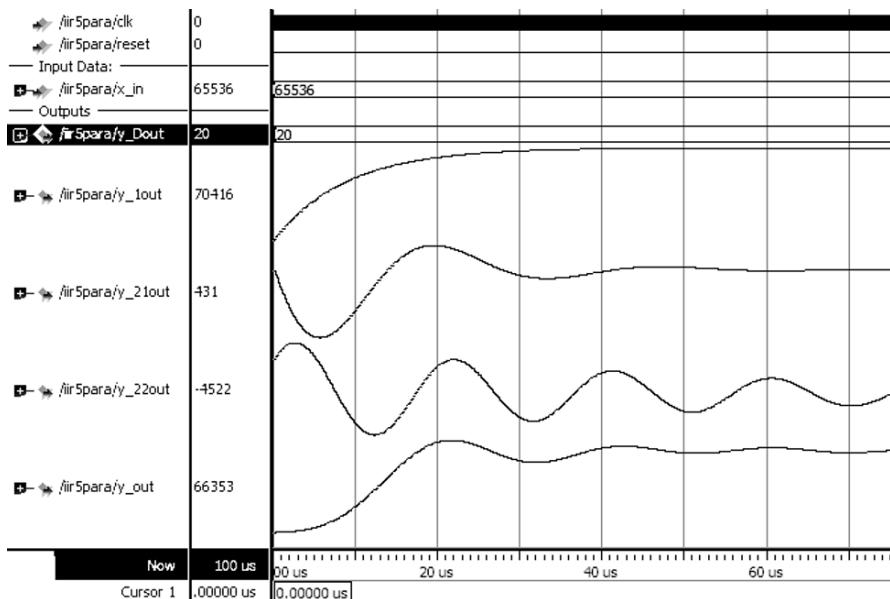


Fig. 4.37. Step response for MODELSIM simulation of the fifth order parallel IIR filter using the modified BiQuad

shown in Fig. 4.37, agrees with the MATLAB simulated results presented in Fig. 4.35a (p. 267) and Fig. 4.23b (p. 253). The 60 μ s simulation requires 600 clock cycles for a 100-ns system clock. The step response maximum occurs after about 215 clock cycles.

4.8

Compared with the direct form implementation the parallel implementation just presented is better in size and speed. It runs about twice as fast, needs only one quarter the LEs and one half the embedded multipliers.

4.5.4 Lattice Design of Narrow Band IIR Filter

The direct form filter, or parallel, or cascade form usually allow an easy transition between the systems difference equation in the z -domain, their pole/zero diagram, or impulse response. This is not that simple for lattice filters and the WDF discussed in this and the next section, respectively. However, there are benefits when designing lattice (or WDF) filters – one is the low coefficient sensitivity to quantization. Another property not used here is that the stability of a lattice filter can easily be verified (all lattice filter coefficients amplitudes must be less than one) which comes in handy when filter coefficients change such as in the case of adaptive filters.

We assume we have already computed poles and zeros (or z -domain transfer function) of the filter we want to build. IIR lattice filters are usually de-

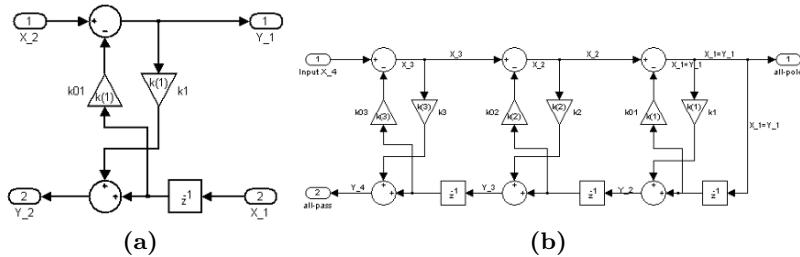


Fig. 4.38. Lattice filter (a) single section (b) three section all-pass/all-pole type

signed in two steps: first the poles of the filter are computed and then different intermediate feedback outputs are combined to implement the feedforward coefficients. To understand better the steps involved let us assume we have the data of a third order filter available and now want to determine the lattice filter parameters for a filter with the same impulse response. To this end let us have a look at a signal section of the lattice shown in Fig. 4.38a. Using the I/O names as in Fig. 4.38a we can now compute the cascade matrix A as follows:

$$\begin{bmatrix} X_{n+1} \\ Y_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & k(n)z^{-1} \\ k(n) & z^{-1} \end{bmatrix} \begin{bmatrix} Y_n \\ X_n \end{bmatrix} \quad (4.29)$$

For a third order filter we need to combine three of these sections and we also have the condition $X_1 = Y_1$; see Fig. 4.38b. The transfer function of the overall system now depends on the definition of the output ports. If we use Y_1 as output then $H(z) = Y_1/X_4$ would become an *all-pole* filter. If we use $H(z) = Y_4/X_4$ then we would have built an *all-pass* filter. To verify the all-pole we can use the following short MATLAB script that uses the **symbolic** toolbox:

```

syms k1 k2 k3 z
disp('Two-pair matrix description:')
A1=[1 k1*z^-1;k1 z^-1];
A2=[1 k2*z^-1;k2 z^-1];
A3=[1 k3*z^-1;k3 z^-1];
disp('Third order transfer X(z)/Y(x):')
P=A3*A2*A1
S=P(1,1)+P(1,2);
collect(S,z)

```

In the first line the symbols are defined. Then the three matrices are built and the product P of the cascade is computed. Since we have $X_1 = Y_1$, the sum $P(1,1) + P(1,2)$ is X_4/Y_1 and the reciprocal the desired quotient $H(z) = Y_1/X_4$.

If we run the MATLAB script we would get as the result:

Third order transfer $X(z)/Y(x)$:

$$\begin{aligned} P = \\ [1+k_3/z*k_2+(k_2/z+k_3/z^2)*k_1, \\ (1+k_3/z*k_2)*k_1/z+(k_2/z+k_3/z^2)/z] \\ [k_3+k_2/z+(k_3/z*k_2+1/z^2)*k_1, \\ (k_3+k_2/z)*k_1/z+(k_3/z*k_2+1/z^2)/z] \end{aligned}$$

$$\text{ans} = 1 + (k_3*k_2+k_1+k_2*k_1)/z + (k_3*k_2*k_1+k_2+k_3*k_1)/z^2 + k_3/z^3$$

or using negative z^{-n} we get

$$H(z) = \frac{1}{1 + (k_3k_2 + k_1 + k_2k_1)z^{-1} + (k_3k_2k_1 + k_2 + k_3k_1)z^{-2} + k_3z^{-3}} \quad (4.30)$$

to transfer this representation back to the standard all-pole polynomial such that we can use it as pole/zero plot or the transfer function via `freqz`; we would compute

$$\begin{aligned} a_4 &= k_3 \\ a_3 &= k_3k_2k_1 + k_2 + k_3k_1 \\ a_2 &= k_3k_2 + k_1 + k_2k_1. \end{aligned}$$

From the symmetry in the matrix P above we can also see that $H(z) = Y_4/X_4$ will become an all-pass filter. To prove this we compute

$$P(2,1)+P(2,2)$$

and we get the reciprocal polynomial

$$k_3+k_2/z+(k_3/z*k_2+1/z^2)*k_1+(k_3+k_2/z)*k_1/z+(k_3/z*k_2+1/z^2)/z$$

i.e., Numerator and denominator are mirror versions, $\text{Num}(z^{-1}) = \text{Den}(z)z^L$, and pole/zeros become mirror versions regarding the unit circle.

Although all-pole and all-pass filters are useful filters by themselves and are indeed used in many applications such as speech processing, that was not our goal this time. To make this lattice filter a universal IIR filter we need to combine different tap outputs to build the feedforward coefficients. A key paper by Gray and Markel [112] shows the details of this computation. To build the complete IIR filter we need then to compute

$$Y(z) = \sum_{m=1}^{M+1} v(m)Y_m(z)X(z), \quad (4.31)$$

i.e., the individual outputs $Y_m(z)$ are weighted and summed to the filter output $Y(z)$; see Fig. 4.39. The individual $Y_m(z)$ and the filter weight $v(m)$ are usually recursive computed via the two equations

$$\begin{aligned} Y_{m+1} &= Y_m z^{-1} + k(m)X_m \\ X_{m+1} - k(m)z^{-1}Y_m &= X_m \end{aligned}$$

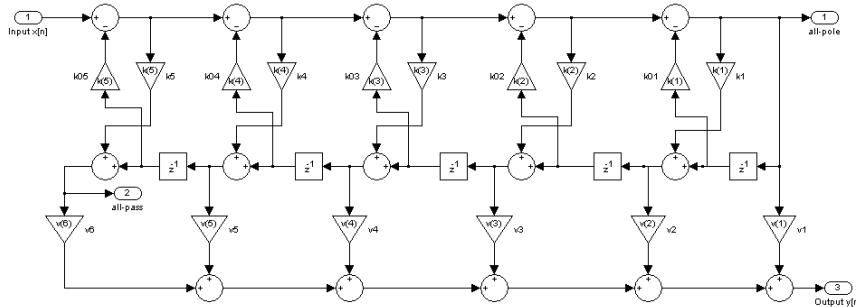


Fig. 4.39. The fifth order IIR lattice filter

This method has been summarized in the MATLAB function `[k, v] = tf2latc` that produces the lattice k parameter as well as the feedforward coefficients v . Using the data from our fifth order filter discussed earlier we get

```

k(1) = -0.999690100
k(2) = 0.999875949
k(3) = -0.999815189
k(4) = 0.999660865
k(5) = -0.970108307

v(1) = 0.000000004445
v(2) = 0.000000009318
v(3) = 0.000003644655
v(4) = 0.000005053596
v(5) = 0.000599705718
v(6) = 0.000303581790

```

Already by looking at the lattice filter parameter we can make some general observations about the characteristic of narrow band IIR lattice filters. From the k parameter we see that the filter is stable, but the values are very close to ± 1 , which indicates that the poles are very close to the unit circle. From the very small feedforward factors v we may already suspect that the lattice filter will produce some substantial amplitudes for the narrow band filter. Let us design the filter and have a look at the integer and fractional requirement of the filter.

The required fractional bits are computed by the same procedure we used for the direct form filter, i.e., each coefficient is quantized and then we measure whether the tolerance scheme of the transfer function is matched. As can be seen from Fig. 4.40, 13 bits are not enough but 14 bits are required to match the tolerance scheme. This is larger as expected, but keep in mind that the feedforward coefficients variation (a.k.a. coefficient spread) is large: $\max(v)/\min(v) = 1.3 \times 10^5$ or 17.1 bits.

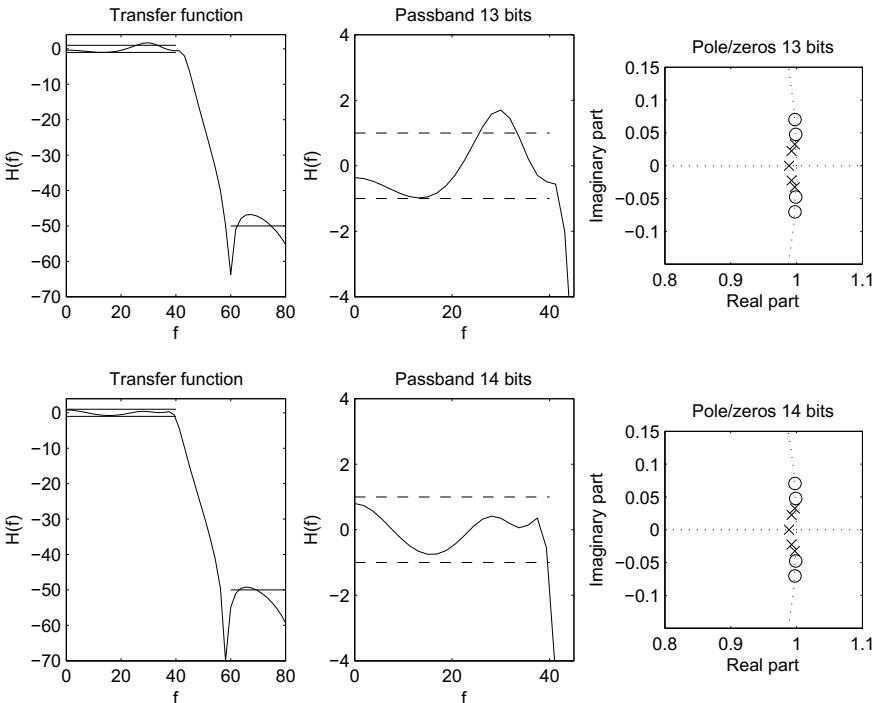


Fig. 4.40. Transfer functions and pole zero plots for quantized coefficients of the fifth order lattice filter

To determine the required integer bits of the filter we use the same procedure as for the direct form filter, i.e., the filter is designed in SIMULINK and then the step-response is measured; see Fig. 4.41. All-pass output and the IIR filter output $y[n]$ are well scaled. However, the worst case gain is the first section output (i.e., the all-pole output) shown in Fig. 4.41b. Worst case $\lceil \log_2(2 \times 10^8) \rceil = [27.6] = 28$ bit internal integer precision is required to avoid overflow in arithmetic. Combining fractional and integer bit width together we see that the lattice filters have substantial bit width, i.e., $14 + 28 = 42$ bits.

Improvements in integer gain are possible if we use the one-multiplier architecture as suggested by Gray and Markel [112]; see Fig. 4.42. These modified sections have the additional benefit that they only need one multiplier. In addition the two different versions allow tuning of the gain between the sections. First let us have a look at the cascade matrix \mathbf{A} of the modified sections. The output of the top left adder is $X_2 - X_1z^{-1}$ and we get for the two output signals

$$Y_1 = X_2 + k_1(X_2 - X_1z^{-1}) \quad (4.32)$$

$$Y_2 = X_1z^{-1} + k_1(X_2 - X_1z^{-1}). \quad (4.33)$$

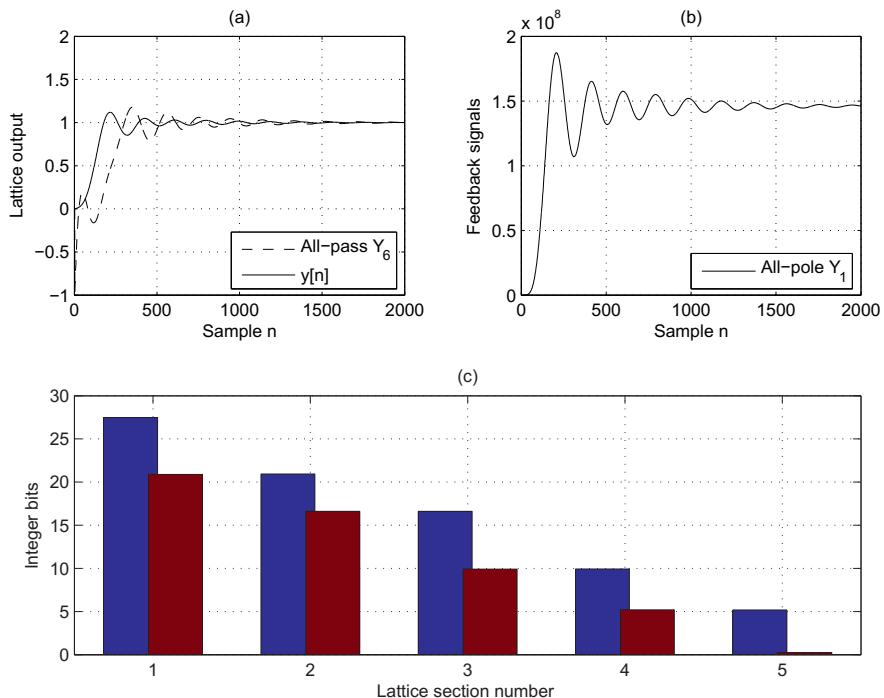


Fig. 4.41. Integer bit width computation for the fifth order lattice filter

For the \mathbf{A} matrix we need the index two variable on the left and the index one on the right, i.e., we rearrange (4.32) as

$$X_2 = \frac{1}{1 + k_1} (Y_1 + k_1 z^{-1} X_1). \quad (4.34)$$

If we now substitute (4.34) into (4.33) it follows that

$$\begin{aligned} Y_2 &= z^{-1} X_1 (1 - k_1) + k_1 \frac{1}{1 + k_1} (Y_1 + k_1 z^{-1} X_1) \\ &= \frac{1}{1 + k_1} (k_1 Y_1 + z^{-1} X_1) \end{aligned} \quad (4.35)$$

The two equations (4.34) and (4.35) can now be combined to produce the general matrix equation

$$\begin{bmatrix} X_{n+1} \\ Y_{n+1} \end{bmatrix} = \frac{1}{1 + k(n)} \begin{bmatrix} 1 & k(n)z^{-1} \\ k(n) & z^{-1} \end{bmatrix} \begin{bmatrix} Y_n \\ X_n \end{bmatrix}. \quad (4.36)$$

If we now compare the \mathbf{A} matrix of the two multiplier (4.29) versions with (4.36) we see that the only difference is the scale factor $1/(1 + k(m))$, otherwise the matrices are the same. If we repeat the computation for the second multiplier type shown in Fig. 4.42b the matrix still remain the same, and the

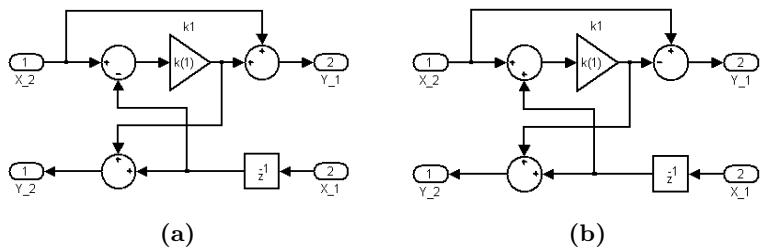


Fig. 4.42. Lattice filter (a) sign is plus (b) sign minus

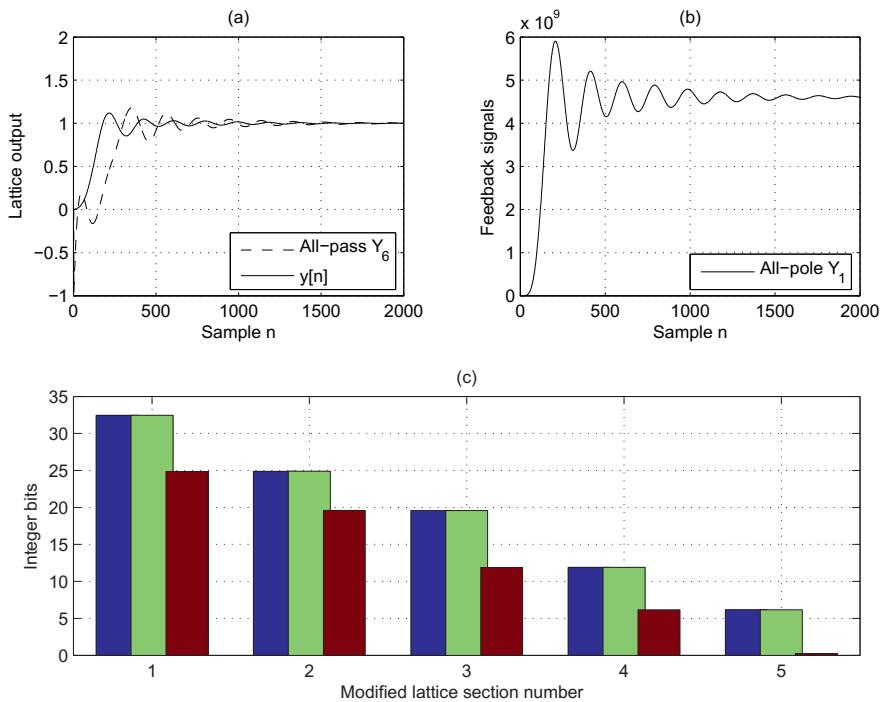


Fig. 4.43. Integer bit width computation for the modified lattice filter using same operation $e = \text{sign}(k) = [-1, 1, -1, 1, -1]$ matching the sign of the k factors

scale factor becomes $1/(1-k(m))$. With these two blocks we now have a tool at hand to control the gain of the blocks. If we like to reduce the gain we set $1/|1 \pm k(m)| > 1$. If we need a gain larger than one we set $1/|1 \pm k(m)| < 1$. This should be a balanced trade off: the gain should not be too large to reduce the internal bit width and it should not become too small otherwise the quantization noise would increase. A brute force method would be to test all $2^5 = 32$ configurations for the fifth order filter. Alternatively, we can use the

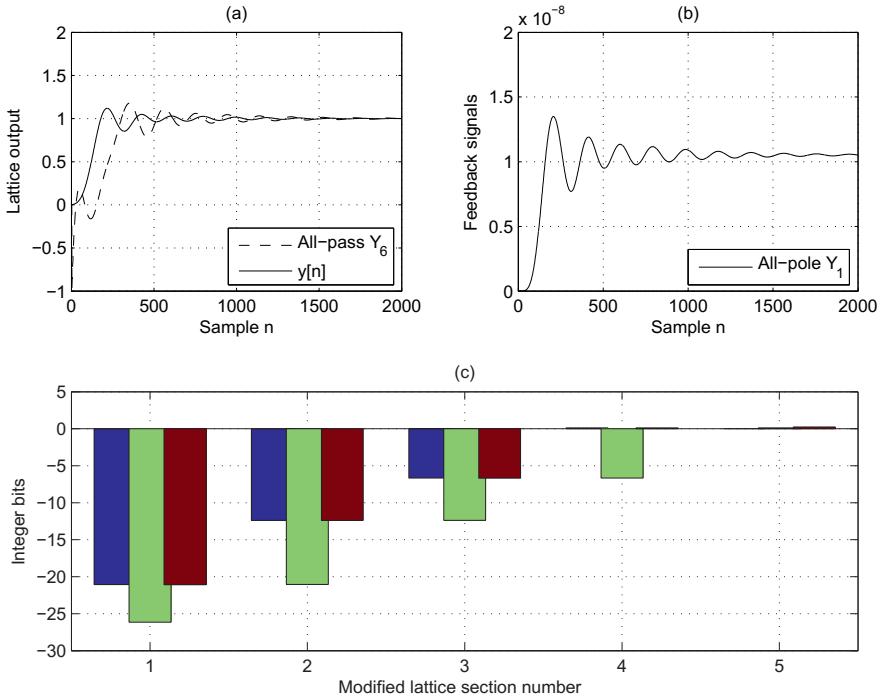


Fig. 4.44. Integer bit width computation for the modified lattice filter using the operation $e = -\text{sign}(k) = [1, -1, 1, -1, 1]$, the opposite to the sign of the k factors

algorithm developed by Gray and Markel [112] that keeps the gain as large as possible but not exceeding an upper limit of one. For our lattice we have all $|k(n)| \approx 1$. If we set the plus/minus operator for $e(n) = \text{sign}(k(n))$ then a maximum gain up to $2^5 = 32$ bit occurs; see Fig. 4.43c. Note from Fig. 4.43b how large the all-pole signal becomes. The bit width grows even larger than for the two multiplier lattices.

At the other extreme we can set $e(n) = -\text{sign}(k(n))$; then the gain is less than one in all stages and only fractional bits are needed; see Fig. 4.44. Note from Fig. 4.44b how small the all-pole signal becomes. If we follow the Gray and Markel [112] method, gains larger and smaller than one are alternating and only the $e(n) = 1$ architecture is used. All signals are now “well” scaled, even the all-pole output, as can be seen from Fig. 4.45b.

Since the gain in each section has changed, it is also necessary to adjust the output weights. It is basically a product of the scale factors $\prod(1 \pm k(m))$. More precisely we need to compute

$$u(m) = v(m)/\pi(m) \quad (4.37)$$

$$\pi(m) = \begin{cases} 1 & \text{for } m = M + 1 \\ \prod_{n=m}^M (1 + e(n)k(n)) & \text{for } m = 1, 2, \dots, M. \end{cases} \quad (4.38)$$

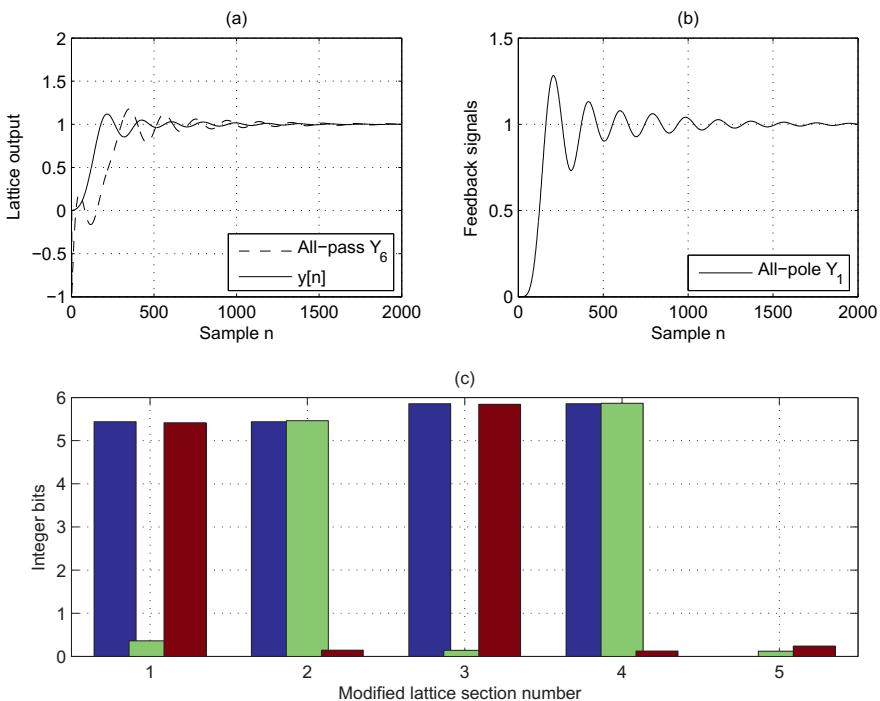


Fig. 4.45. Integer bit width computation for the modified lattice filter using the $e(n) = [1, 1, 1, 1, 1]$ sign operation

A small MATLAB script can be used to compute these modified feedforward coefficients. The k coefficients do not change in the modified lattice architecture. For our fifth order filter with the architecture choice of $e(n) = [1, 1, 1, 1, 1]$ we will get the following values:

$$\begin{aligned} u(1) &= 0.649195217878 \\ u(2) &= 0.000421764763 \\ u(3) &= 0.329929109321 \\ u(4) &= 0.000084546124 \\ u(5) &= 0.020062621320 \\ u(6) &= 0.000303581790 \end{aligned}$$

An additional benefit is that the coefficient spread is now also reduced to $\log_2(\max(u)/\min(u)) = 12.9$ bits compared to the 17.1 bits for the v coefficients of the two multiplier architecture.

Although the bit growth problem has been mitigated with the Gray–Markel method, another inherent worst case delay path problem arises with the lattice filter. This can be seen from Fig. 4.39, p. 274. There is a long delay path from the filter input to the output of five adders and one multiplier for

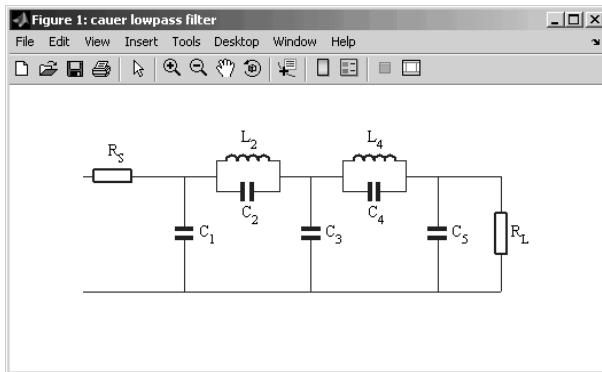


Fig. 4.46. WDF analog fifth order prototype filter

the standard lattice. This is even worse for the Gray–Markel configuration with ten adders and five multipliers. This will substantially reduce the maximum register performance of the lattice filter, making it less attractive for an HDL FPGA design.

4.5.5 Wave Digital Filter Design of Narrow Band IIR Filter

The concept of Wave Digital Filters (WDFs) takes a different approach and WDFs therefore look quite different from most other digital filters. These WDFs are typically derived from normalized analog filters in ladder form using R/L/C analog components (e.g., via look-up tables in filter books). A typical analog fifth order Cauer a.k.a. elliptic filter is shown in Fig. 4.46.

Next the filter is transformed in the digital domain via bilinear transform and component transformations. Then the filters are denormalized to meet the design passband and stop band behavior. A capacitor in the analog domain is translated into a unit delay z^{-1} , while the inductor becomes a negative gain, i.e., $-z^{-1}$. Since the source resistor and load resistor are usually normalized to one, they are omitted in the z -domain. Incoming waves (i.e., signals) are assigned the variable A_k , while outgoing waves of a block are named B_k . To connect the normalized C and L values, *adaptors* are used. Adaptors are used to take care of the type of the connection (i.e., serial or parallel) as well as the adjustment of the normalized component values. Referring to Fig. 4.46 we see that we need a parallel adaptor for the “vertical” components and a serial adaptor for the “horizontal” positioned components. In the following we will discuss such serial and parallel adaptors from a system equation and SIMULINK implementation standpoint.

Figure 4.47b shows the symbol of a parallel three-port adaptor, while Fig. 4.47a shows the SIMULINK design. Note that later in the hardware design the general multiplication can be replaced by a constant coefficient multiplier.

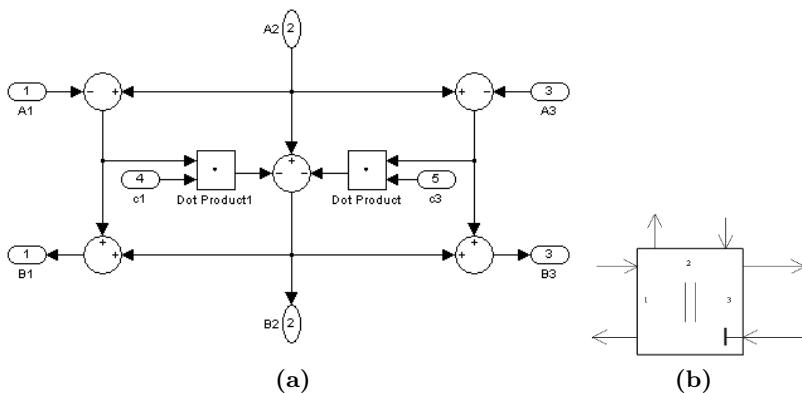


Fig. 4.47. WDF parallel three-port adaptor (a) SIMULINK model. (b) Symbol

However, to determine the required coefficient quantization the use of a general multiplier is more convenient.

In the literature the symbols have changed over time. The symbols we use follow closely the important review paper from 1986 by A. Fettweis [101]. The I/O transfer matrix for the parallel adaptor is computed from the Kirchhoff laws, i.e., that the sum of the current is zero and a voltage loop must also be zero; for details see for instance [117, Chap. 9]. For a three-port adaptor we get

$$\begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} c_1 & 2 - c_1 - c_3 & c_3 \\ c_1 & 1 - c_1 - c_3 & c_3 \\ c_1 & 2 - c_1 - c_3 & c_3 - 1 \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \\ A_3 \end{bmatrix}. \quad (4.39)$$

In order to avoid delay free loops we will also require three-port adaptors such that one port is reflection free, i.e., the output wave B_k does not depend on the input wave A_k of that port, or in short $B_k \neq f(A_k)$. In WDF terms such ports are called *matched* ports and from inspection of (4.39) we see that, if we desire $B_3 \neq f(A_3)$, that requires $c_3 = 1$. Then (4.39) for the matched three-port adaptor simplifies to

$$\begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} c_1 & 1 - c_1 & 1 \\ c_1 & -c_1 & 1 \\ c_1 & 1 - c_1 & 0 \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \\ A_3 \end{bmatrix}. \quad (4.40)$$

The simplified SIMULINK design and the circuit symbol are shown in Fig. 4.48.

For the serial three-port adaptor similar equations and signal flow graphs can be derived. Figure 4.49a shows the serial adaptor circuit and Fig. 4.49b the circuit symbol. The matrix equation for the serial adaptor becomes:

$$\begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} 1 - c_1 & -c_1 & -c_1 \\ -c_3 & c_1 + c_3 - 1 & c_1 + c_3 - 2 \\ c_1 + c_3 - 2 & -c_3 & 1 - c_3 \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \\ A_3 \end{bmatrix}. \quad (4.41)$$

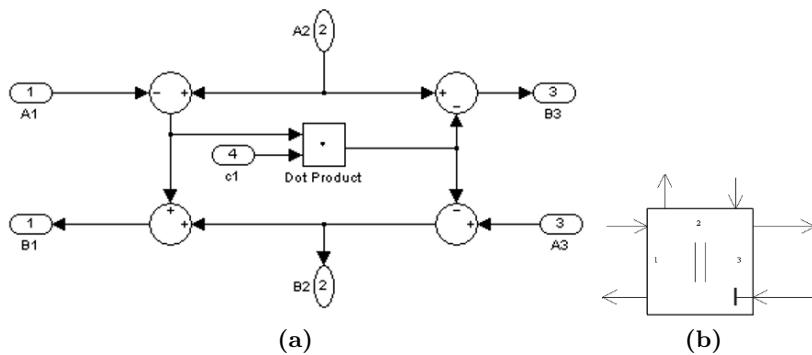


Fig. 4.48. WDF parallel matched three-port adaptor. (a) SIMULINK model. (b) Symbol

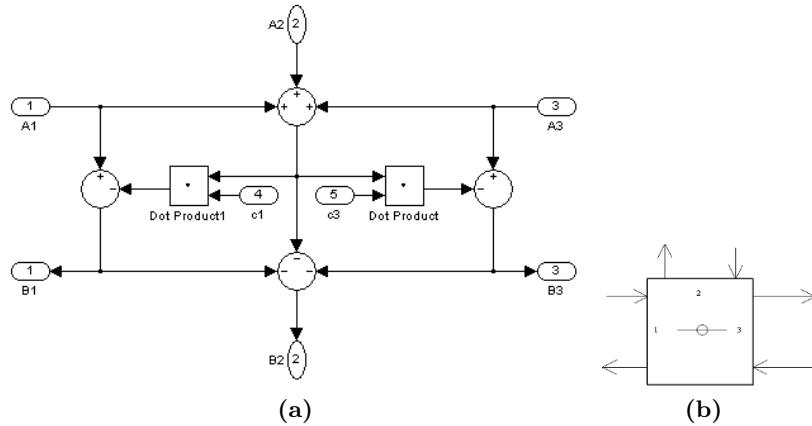


Fig. 4.49. WDF serial three-port adaptor. (a) SIMULINK model. (b) Symbol

For the serial adaptor we need a delay free loop version, i.e., a match configuration such that the third port is reflection free, i.e., $B_3 \neq f(A_3)$. Again this is true for $c_3 = 1$ and the three-port equation (4.41) simplifies to

$$\begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} 1 - c_1 & -c_1 & -c_1 \\ -1 & c_1 & c_1 - 1 \\ c_1 - 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \\ A_3 \end{bmatrix}. \quad (4.42)$$

The delay free version is beneficial to the hardware implementation because now one multiplier less need to be implemented. The simplified SIMULINK design and the circuit symbol for the matched three-port serial adaptor is shown in Fig. 4.50.

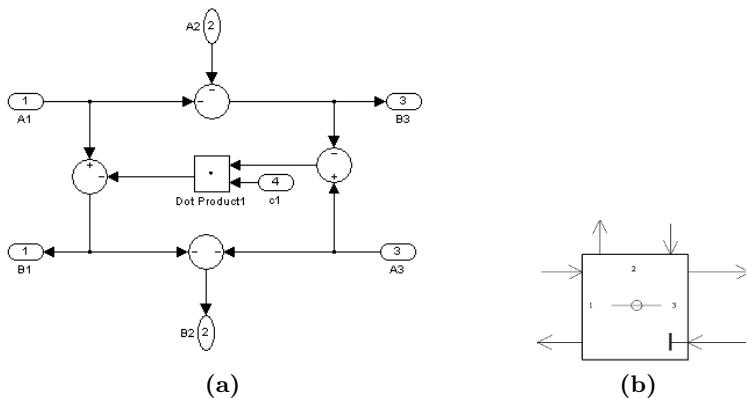


Fig. 4.50. WDF serial matched three-port adaptor. (a) SIMULINK model. (b) Symbol

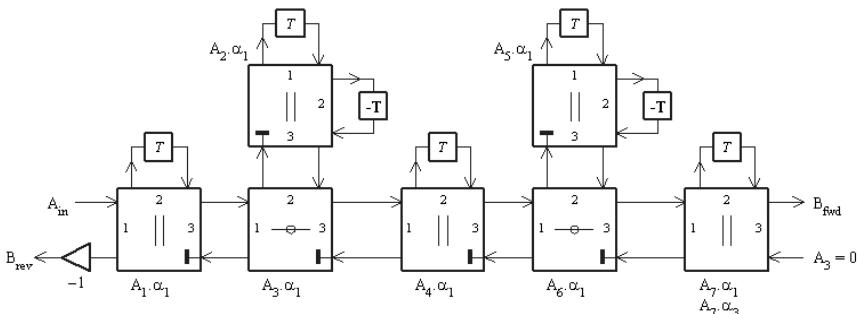


Fig. 4.51. Fifth order WDF digital filter

Having developed the parallel and serial adaptors and their matched versions we now have all the tools to translate the circuit architecture from our initial analog design of Fig. 4.46 (p. 280) into the digital domain and the corresponding digital WDF is shown in Fig. 4.51.

What we do not have so far are the digital filter coefficients for the WDF. There is an excellent public domain WDF toolbox developed by H. Lincklaen Arriens from TU Delft that can be used to generate the filter coefficients (and architectures) using a few lines of MATLAB code. Here is the code and parameter for the functions in use for the fifth order filter with $F_{\text{pass}} = 40 \text{ Hz}$, $F_{\text{stop}} = 60 \text{ Hz}$, $F_s = 7680 \text{ Hz}$, $A_{\text{pass}} = 1 \text{ db}$, and $A_{\text{stop}} = 48 \text{ dB}$:

```
%> Generate the filter coefficients for system:
%%X = NLP_LADDER('cauer',filterOrder,passBandRipple_dB, ...
%%                      stopBandRipple_dB,skwirNorm,freqNormMode)
Ladder = nlp_ladder('cauer',5,1,48,'a',1);
```

```

%% Denormalize the ladder filter
%% LpLadder = NLADDER2LP(NlpLadder,cutOffFrequency)
dLadder=nladder2lp(Ladder,fz2fs(40/7680))

%% Compute the WDF filter data and impulse responses:
%%      [WDF,fwdB,revB,allB] =
%%      LADDER2WDF(Ladder,wdfType,impulseResponseLength,figNo)
%% also returns forward output, reverse output, all B-out-
%% puts in a 3 column by 'numbers of adaptors' matrix form.
[WDF,fwdB,revB,allB]=ladder2WDF(dLadder,'3p',8*512,3);

```

and the MATLAB result is

```

Configuration 1:
Rs    1.00000 Ohm
C01   1.99211 F   in shunt arm
L02   0.97767 H, parallel with
C02   0.23108 F   in series arm
C03   2.46127 F   in shunt arm
L04   0.76116 H, parallel with
C04   0.64686 F   in series arm
C05   1.68564 F   in shunt arm
RL    1.00000 Ohm

Configuration 2:
Rs    1.00000 Ohm
C01   1.68564 F   in shunt arm
L02   0.76116 H, parallel with
C02   0.64686 F   in series arm
C03   2.46127 F   in shunt arm
L04   0.97767 H, parallel with
C04   0.23108 F   in series arm
C05   1.99211 F   in shunt arm
RL    1.00000 Ohm

dLadder =

```

```

elements: 'rCpCpCR'
values: [9x2 double]

Adaptor 1: 3p parallel, p3 matched : alpha1 = 0.00815
Adaptor 2: 3p parallel, p3 matched : alpha1 = 0.99882
Adaptor 3: 3p serial,  p3 matched : alpha1 = 0.10329
Adaptor 4: 3p parallel, p3 matched : alpha1 = 0.07774
Adaptor 5: 3p parallel, p3 matched : alpha1 = 0.99946

```

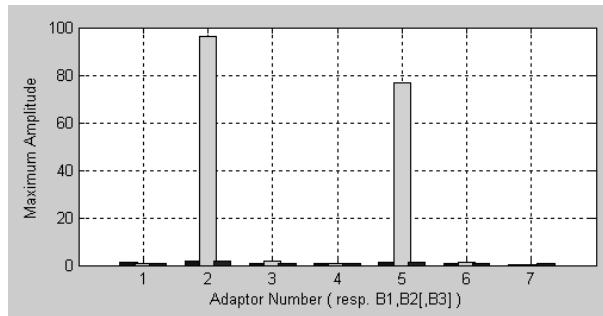


Fig. 4.52. WDF maximum amplitude of all ports of the fifth order filter. (Snapshot from the WDF toolbox by H. Lincklaen Arriens)

```

Adaptor 6: 3p serial,   p3 matched : alpha1 = 0.19518
Adaptor 7: 3p parallel      : alpha1 = 0.46866
                           alpha3 = 0.01472
Brev needs negation

```

The toolbox not only computes the analog and digital circuits; it also gives an estimate via the inverse FFT or the impulse response of the worst case growth of each port variable; see Fig. 4.52. We can see that only the matched parallel adaptor 2 and 5 show substantial integer width. With a maximum amplitude of around 100 an integer bit width of $\lceil \log_2(100) \rceil = 7$ bits seemed sufficient. All other adaptors should be designed with just 1 or 2 (guard) integer bits.

WDFs are known to behave well in the presence of coefficient quantization. Remember from our discussion earlier that we have used general multipliers instead of fixed coefficient values. This enable us to determine interactively the required precision N of the coefficients by setting $c_q(i) = \text{round}(c(i) \times 2^N) / 2^N, \forall i$, i.e., we first multiply each coefficient by a power-of-two, then quantize it and then divide by 2^N since all coefficients are assumed to be fractional only. We try to find the smallest N such that the transfer function of the filter still fits in the desired tolerance scheme for our filter. The simulation result in Fig. 4.53 shows that 14-bit coefficient quantization is sufficient to meet passband and stopband requirements.

Overall the WDF shows good integer gain behavior and low component count. However, the major disadvantage of the direct implementation of the WDF is that we have a direct path from input to output of the filter. The “wave” travels all the way from the input to the output adaptors for the forward output. Unfortunately, even without using the complementary output, the backward path has to be calculated for feeding all delay elements before a next input sample arrives. If an implementation with parallel executing hardware can be designed, both forward and backward paths can be strongly reduced by using a “symmetric” WDF, i.e., one with the unmatched

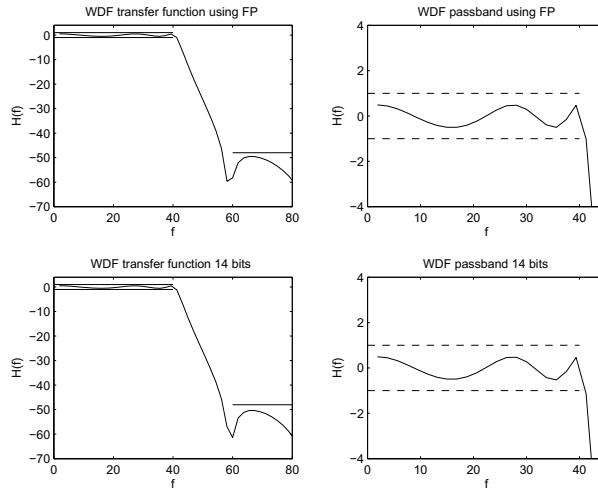


Fig. 4.53. Transfer functions plots for quantized coefficients of the wave digital filter using floating-point (FP) precision (*upper row*) using 14-bit coefficient quantization (*lower row*)

Table 4.4. Comparison of design options for narrow band IIR filters

Architecture	Operations			Delay	Bits		
	Adds	Mult.	\sum		I	F	\sum
Direct	10	11	21	$2+1\times$	33	30	63
Cascade BiQuad	10	$8+2$	18	$2+1\times$	11	14	25
Cascade Mod. BQ	10	$8+2$	18	$1+1\times$	8	14	22
Parallel BiQuad	7	11	18	$2+1\times$	11	18	29
Parallel Mod. BQ	7	11	18	$1+1\times$	1	18	19
Lattice 2 Mul.	15	16	31	$5+1\times$	28	14	32
Lattice 1 Mul.	20	11	31	$10+5\times$	6	14	20
WDF	29	8	37	$9+3\times$	7	14	21

adaptor located in the middle instead of at the end (only odd order filters)⁶. In [117, p. 359] such a seventh order WDF filter with the unmatched adaptor in the middle is presented. It is difficult to pipeline such a complicated architecture since the pole/zero locations of a pipelined architecture should not change.

For a first overview of the different design choices let us therefore compare the architecture in term of resources used, integer and fractional bit requirements, and the longest path. Without an HDL implementation this data will not be exact, but a good first estimate of the most promising architectures.

⁶ Thanks to H. Lincklaen Arriens for pointing out this improvement [118].

The direct form does not look bad in terms of arithmetic operation, but the filter has a very high bit width requirement. In floating-point arithmetic the double precision (i.e., 64 bit) would be necessary to obtain this high precision. The Cauer filter makes the cascade structure look particularly good in terms of required multipliers since the zeros are on the unit circle. However, if scaling is used to bring the section gain back to one, additional multipliers are required. By using the transposed direct form 1 (i.e., BiQuad) the worst case path also becomes reasonable. With the modified BiQuad in parallel or cascade form we achieve the shortest delay path length with one adder and one multiplier. The parallel form is attractive since it uses fewer adders and the ordering problem as in the cascade implementation does not occur and BiQuads can be verified parallel. Lattice filter are a little high on the arithmetic count compared to the other filters, but have a good fractional bit width. However, the integer bit width is substantial and does not make the lattice a much better choice than the parallel or cascade form. The WDF filter comes in as winner for the multiplier but has a higher adder count. The WDF most likely in these direct forms is not attractive due to the long delay path of the filter. This should improve if we use the Lattice WDF discussed next.

4.6 All-Pass Filter Design of Narrow Band IIR Filter

An all-pass filter is defined as a filter with a transfer function of “one” at all frequencies and is usually used in DSP to generate phase delays. Now you may ask: How can such an all-pass be used to build a narrow band filter if the amplitudes are one for all frequencies? The key idea here is to use two all-pass filters working together. We build a sum or difference of the two outputs. If the two all-pass filters have gain one and phase delay zero in the passband but a differential phase delay of 180° in the stopband then we get $\sin(\omega t + 180^\circ) = -\sin(\omega t)$. If we now add these two all-pass signals then we have realized a stopband at the frequency ω and a narrow filter can be built. Figure 4.54 shows the phase relation of a typical all-pass filter design.

In addition, since $|F(\omega)| = 1$ for the two all-pass filters the integer gain of the filter should stay small, which will be an additional benefit when designing the filter in hardware.

There is however a small constraint we need to follow when we build the two parallel all-pass systems. It turns out that lowpass and highpass filters can only have an odd order [119]. If we try to build even length lowpass or highpass filters the coefficients will become complex with a substantially higher implementation effort if we try to build filters for real data [120]. Bandpass or bandstop filters with real (not complex) coefficients on the other hand require that the order of the overall filter is even.

If we draw the diagram with the two all-pass filters followed by a sum and difference operation it looks like a (single) stage of a lattice; see Fig. 4.38,

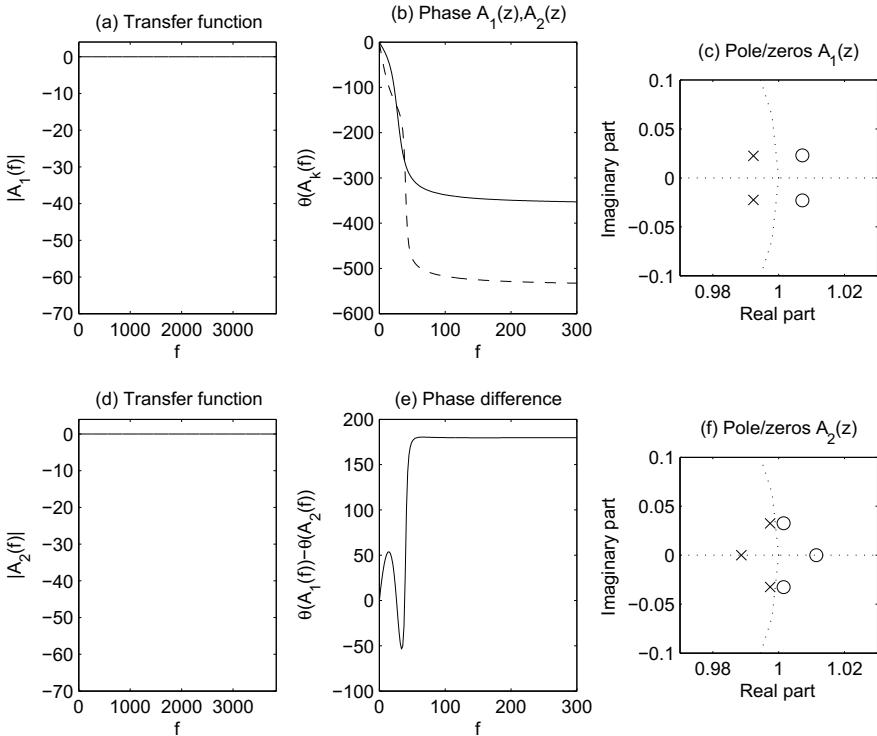


Fig. 4.54. Transfer functions, phase, and pole zero plots for parallel all-pass filter design

p. 272. The idea was first used in combination with WDF and therefore the term lattice WDF or LWDF is used in the literature, although it is only a one stage lattice and the idea not only applies to WDF but any all-pass filter, i.e., cascade or parallel BiQuads or Gray–Markel lattices can be used. The lattice filter with one multiplier seemed to be an especially efficient choice since by default it is an all-pass filter.

A question that may arise is whether we need to redesign our filter from scratch, but Gazsi showed that most popular filters like Butterworth, Chebyshev, or elliptic filters can be transformed into the parallel all-pass form [113]. This property seemed to be related to the fact that the bilinear transform was used to design the coefficients of these filters.

Since the system output is computed via sum or difference of the two all-pass filters, i.e.,

$$H_{1,2} = \frac{1}{2} (A_1(z) \pm A_2(z)), \quad (4.43)$$

we expect that the poles of the all-pass and the original filter match. Gaszi showed that the poles of the two all-pass filters are distributed in an alternate

fashion towards the unit circle. For our fifth order system that means that the real pole and the pole closed to the unit circle are combined in one filter and the other pole pair in the second all-pass, i.e., we have one third order and a second order system in parallel; see Fig. 4.54c,f.

MATLAB supports this filter design with the “transfer function to coupled all-pass lattice” conversion called `tf2cl`. If we apply this function

```
[latt1, latt2] = tf2cl(B,A)
```

to our fifth order filter from the last section we get the output

$$\begin{aligned} \text{latt1} &= -0.9997 \quad 0.9852 \\ \text{latt2} &= -0.9996 \quad 0.9998 \quad -0.9846 \end{aligned} \tag{4.44}$$

In order to plot the transfer function and determine the pole/zero plot we transfer the two lattice back to the direct form

```
[num1,den1] = latc2tf(latt1,'allpass')
[num2,den2] = latc2tf(latt2,'allpass')
```

and we get

$$\begin{array}{llll} \text{num1} = & 0.9852 & -1.9847 & 1.0000 \\ \text{den1} = & 1.0000 & -1.9847 & 0.9852 \\ \text{num2} = & -0.9846 & 2.9682 & -2.9835 \quad 1.0000 \\ \text{den2} = & 1.0000 & -2.9835 & 2.9682 \quad -0.9846 \end{array}$$

Figure 4.54a,d shows the transfer function of the two all-pass filters. The phase of the individual filter is shown in Fig. 4.54b and the difference in Fig. 4.54e. Note that we clearly have a 180° differential phase in the stopband. The poles are alternately assigned to the different all-pass filters as Fig. 4.54c,f shows.

Let us now develop how the four filter choices, i.e., cascade or parallel BiQuads, WDF, and lattice compare in the parallel all-pass configuration in terms of required integer and fractional bits. This is the same comparison we made in the last section, and we can therefore keep it short. We use the modified BiQuad for parallel and cascade design and the one multiplier lattice by Gray–Markel. As WDF all-pass the three-port adaptor offers the best performance.

4.6.1 All-Pass Wave Digital Filter Design of Narrow Band IIR Filter

The MATLAB script for the lattice WDF filter is similar to the standard WDF design if we again use the toolbox from H. Lincklaen Arriens from TU Delft:

```

%% 2- and 3-port circuit design as in Anderson et al (1995)
%% Generate the filter coefficients for system:
%% [Hs,wp] = HS_CAUER(filterOrder,passBandRipple_dB, ...
%% stopBandRipple_dB,skwirMode,cutOffFrequency,freqNormMode)
Hs = Hs_cauer(5,1,50,'a',1,1)

%% NLP2LP      Normalized lowpass to lowpass transformation
dHs = nlp2lp(Hs,fz2fs(40.75/7680));

%% Calculates the coefficients of a Lattice WDF
%% [LWDF,Hz,Messages] = HS2LWDF(Hs,figNo)
[LWDF, Hz, Messages] = Hs2LWDF(dHs,3)
%showLWDF
Hz2=LWDF2Hz(LWDF); plotHz(Hz2,1);

```

This script will plot the transfer function of the LWDF and the basic architecture for the two-port configuration as shown in Fig. 4.55. Note that the original specification (passband 40 Hz → 40.75 Hz; stopband ripple 48 dB → 50 dB) was modified such that the quantized filter still meets the required pass- and stopband specification. The lattice WDF are designed such that the high- and lowpass are power complementary and meet at the 3 dB mark. It turns out that for our narrow band filter specification with ± 1 ripple the passband will roll off too early if we specify a 40 Hz passband and we need to increase the passband just a little to have a 40 Hz passband [118].

In order to quantize the multiplier factors we need to extract these from the LWDF.

```

%% Upper 1.order
beta(1)=LWDF.gamma(1,1,1);

%% Upper 2.order
beta(2)=LWDF.gamma(1,2,1);
beta(3)=LWDF.gamma(2,2,1);

%% Lower 2. order
beta(4)=LWDF.gamma(1,1,2);
beta(5)=LWDF.gamma(2,1,2);

```

and the values for the two-port coefficient are

$$0.9887 \quad -0.9959 \quad 0.9995 \quad -0.9853 \quad 0.9997 \quad (4.45)$$

The toolbox does not provide the three-port realization, but we can use the three-port realization from Anderson/Summerfield/Lawson [121] to make the transformation without much effort. Let us call these coefficients now β_k to distinguish them from the three-port coefficients where we use the γ_k . The transfer function of the two-port adaptor is given by

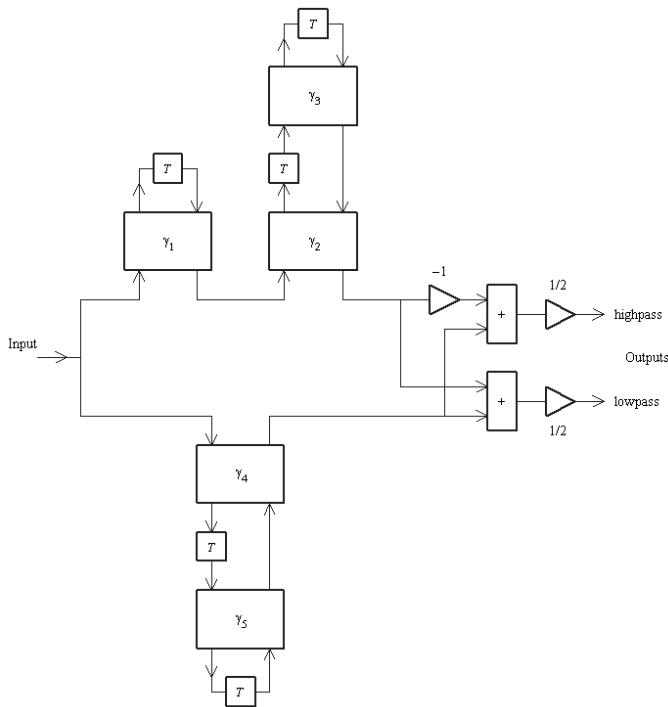


Fig. 4.55. LWDF fifth order digital filter using two-port adaptors

$$h_2(z) = \frac{-\beta_1 - \beta_2(1 - \beta_1)z^{-1} + z^{-2}}{1 - \beta_2(1 - \beta_1)z^{-1} - \beta_1z^{-2}} \quad (4.46)$$

If we now compare $h_2(z)$ with the z transfer function for the three-port design

$$h_3(z) = \frac{-\gamma_1 - \gamma_2 - 1 + (\gamma_2 - \gamma_1)z^{-1} + z^{-2}}{1 + (\gamma_2 - \gamma_1)z^{-1} + (-\gamma_1 - \gamma_2 - 1)z^{-2}} \quad (4.47)$$

we can see that

$$\gamma_1 = -\frac{1}{2}(1 - \beta_1)(1 - \beta_2) \quad (4.48)$$

$$\gamma_2 = -\frac{1}{2}(1 - \beta_1)(1 + \beta_2) \quad (4.49)$$

Now we can recalculate for three-port adaptors the multiplier coefficients via

```
%% Upper 1.order
gamma(1) = beta(1);

%% Upper 2. order
gamma(2) = -0.5*(1-beta(2))*(1-beta(3));
```

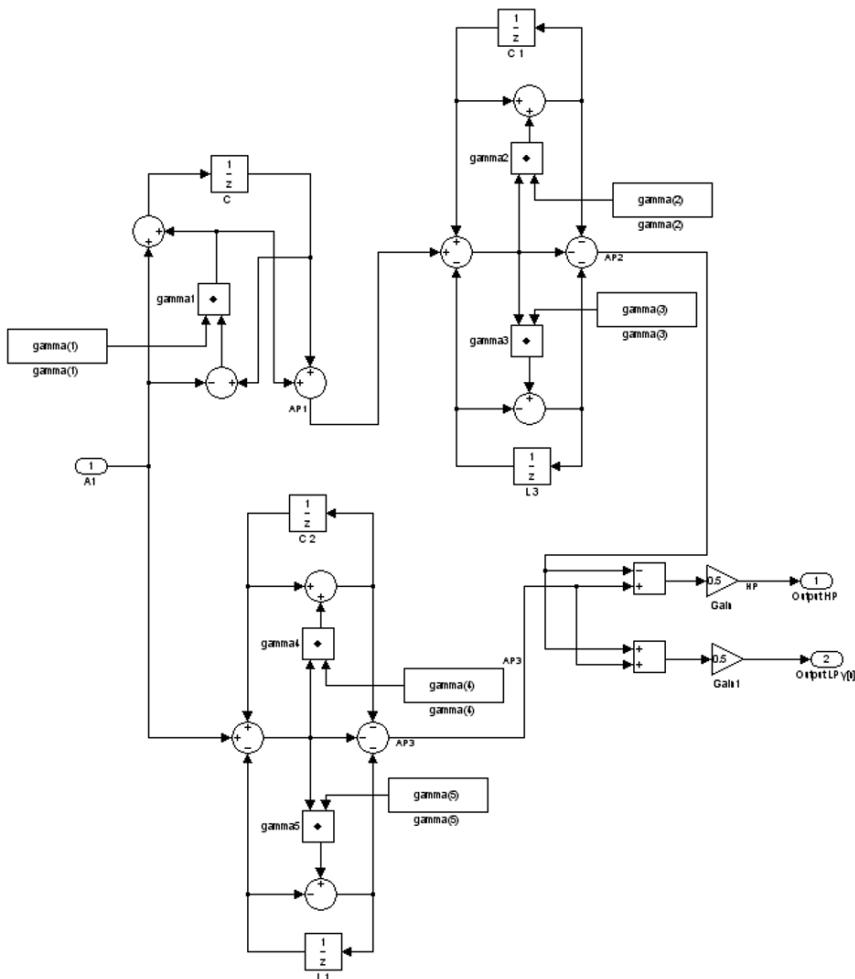


Fig. 4.56. LWDF fifth order digital filter using three-port adaptors

$$\gamma(3) = -0.5 * (1 - \beta(2)) * (1 + \beta(3));$$

% Lower 2. order

$$\gamma(4) = -0.5 * (1 - \beta(4)) * (1 - \beta(5));$$

$$\gamma(5) = -0.5 * (1 - \beta(4)) * (1 + \beta(5));$$

and finally get the following γ values for the three-port configuration:

$$0.988727 - 0.000528 - 1.995400 - 0.000282 - 1.985024 \quad (4.50)$$

The architecture for the three-port LWDF filter is shown in Fig. 4.56.

The last task before the filter can be realized is to have a closer look at the required bits for integer and fractions. A good starting point would be

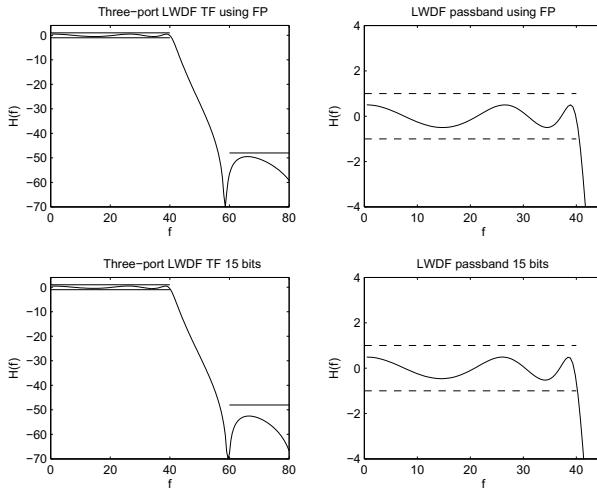


Fig. 4.57. Transfer functions (TFs) plots for quantized coefficients of the lattice wave digital filter using three-port adaptors using floating-point (FP) precision (upper row) using 15 bit coefficient quantization (lower row)

the WDF filter discussed in the last section. It turns out that a 15 bit fraction is enough for both two-port and three-port filters; see Fig. 4.57.

The integer bits are computed by carefully monitoring the adder outputs of the design and reporting the (absolute) maximum values. The results are shown in Fig. 4.58. The maximum stays for both configurations under the value 64 and a 6-bit integer precision is sufficient.

The resource and delay estimates do improve when compared with the standard WDF. The two-port design require 16 adders and five multipliers, that include one adder for the lowpass lattice output. If the highpass lattice output is needed too then just one more adder is needed. For the three-port configuration the same number of multipliers are needed, however the number of adders is reduced to 12. Compared to the 29 adders and eight multipliers of the standard WDF, the LWDF give about 50% hardware reduction. An even bigger advantage however comes in terms of delay. The LWDF has no feedback path in the two- or three-port design and we can introduce pipeline register after each two- or three-port section, reducing the delay to two adders and one multiplier for the two-port and three adders and one multiplier for the three-port design. We just need to make sure that the addition pipeline delays of the upper and lower path match and we would add two additional registers in both branches.

4.6.2 All-Pass Lattice Design of Narrow Band IIR Filter

The basic feature of just the lattice filter were discussed in Sect. 4.5.4 and it turns out that the basic lattice has an all-pole and all-pass output; see

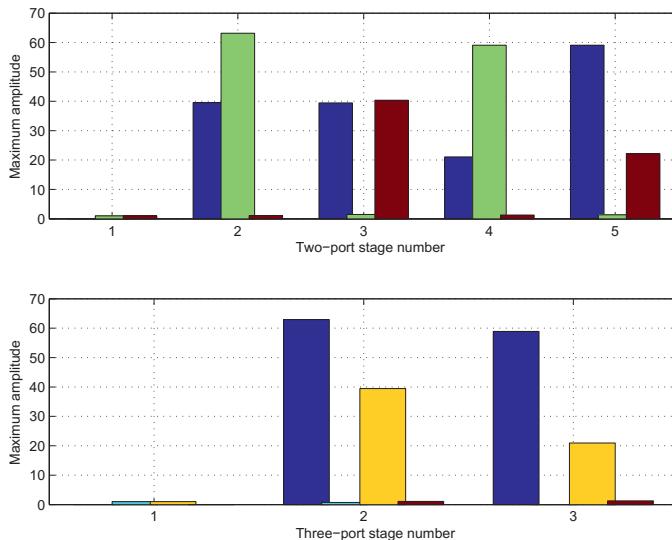


Fig. 4.58. Maximum amplitude computation for the wave digital filter using the two-port configuration (*upper row*) using the three-port configuration (*lower row*)

Fig. 4.38, p. 272. For a lattice filter an all-pass design task can be considered an easy task since by design the lattice has an all-pass output without the need of any output summation as used in the general IIR filter design. The overall arithmetic count of the two parallel all-pass designs is therefore substantially improved compared to the standard single lattice design. The remaining concern for a narrow band filter is the large integer gain usually associated with the standard two multipliers per section lattice architecture. The use of the Gray–Markel one multiplier per lattice section allows a more precise tuning of the gain within the lattice architecture reducing the integer bit requirement from 16 for the two multiplier configuration to six integer bits for the one multiplier per section architecture. The longest delay path also improves to three adders and one multiplier. The coefficient had be computed earlier in (4.44), p. 289. For the one multiplier architecture the Gray and Markel optimization give $(-, -)$ sign for the second order and $(+, +, +)$ for the third order.

4.6.3 All-Pass Direct Form Design of Narrow Band Filter

An implementation of the all-pass filters direct form implementation can benefit from the fact that numerator and denominator are mirror polynomials that generate pole/zero locations that are symmetric to the unit circle. Due to stability requirements the poles are inside the unit circle and the zeros are at the mirror location outside the unit circle. This fact can be used in the transposed direct form 1 with one large block multiplier for all coefficient to

reduce the number of required multipliers by 50%. As a result these architectures usually have the lowest arithmetic count, i.e., 16 for our fifth order Cauer filter. However, by taking advantage of these coefficient similarities we lose the option to do a “zero first” design as in our modified BiQuad designs; see Fig. 4.31, p. 263. As a consequence the overall required integer bit width of the direct form implementation is the highest at 12 bits. The fractional precision with a direct architecture is also substantial with 22 bits. Overall the bit width requirement is reduced from 63 bits in the one filter architecture to 34 bits for the two all-pass filter design, but still substantially higher than for the other filter architectures.

4.6.4 All-Pass Cascade BiQuad of Narrow Band Filter

The direct form can be slightly improved if we use a BiQuad configuration instead of the direct form. Since our system is small the upper section of second order cannot be improved, but the lower section of third order can be implemented with one BiQuad and another first order system. This will reduce the required fractional bit width to 16, and the overall bit width to 28. The other parameters like hardware effort and delay will stay the same.

Table 4.5. Comparison of lattice all-pass design options for narrow band IIR filters

Architecture	Operations			Delay	Bits		
	Adds	Mult.	\sum		I	F	\sum
Direct	11	5	16	$2+1\times$	12	22	34
Cascade BiQuad	11	5	16	$2+1\times$	12	16	28
Parallel BiQuad	10	9	19	$2+1\times$	12	16	28
Lattice 2 Mul.	11	10	21	$3+1\times$	16	14	30
Lattice 1 Mul.	16	5	21	$6+3\times$	7	14	21
LWDF 2 port	16	5	21	$2+1\times$	6	15	21
LWDF 3 port	12	5	17	$3+1\times$	6	15	21

4.6.5 All-Pass Parallel BiQuad of Narrow Band Filter

The parallel form was our favorite direct implementation if we do not use the all-pass configuration. However, for the all-pass design the parallel configuration has the major disadvantage that if we try to implement the filter in parallel form the coefficient symmetry is lost. Let us demonstrate this for the lower third order all-pass with

$$\begin{aligned} \text{num2} &= -0.9846 & 2.9682 & -2.9835 & 1.0000 \\ \text{den2} &= 1.0000 & -2.9835 & 2.9682 & -0.9846 \end{aligned} \quad (4.51)$$

since as the upper all-pass is second order it does not need to be modified. We first decompose the lower system into single pole sum via

```
[R,P,D] = residuez(num2,den2).
```

The two complex poles should be combined to a real second order system and the first order system put into the usual numerator/denominator form

```
[b1,a1] = residuez([R(1) R(2)], [P(1) P(2)], 0);
[b2,a2] = residuez(R(3), P(3), 0);
```

and we get the following values for our parallel BiQuad implementation of the all-pass:

$$\begin{aligned} D &= -1.0156 \\ b1 &= 0.0063 \quad -0.0065 \quad 0 \\ a1 &= 1.0000 \quad -1.9948 \quad 0.9959 \\ b2 &= 0.0246 \quad 0 \\ a2 &= 1.0000 \quad -0.9887 \end{aligned}$$

We notice now that the parallel filter no longer has the same coefficients as the original all-pass from (4.51). We should therefore expect a higher implementation effort for the parallel implementation of the all-pass than for the direct or cascade forms, i.e., 19 operations compared to 16 for the cascade BiQuad. The delay should be the same and integer and fractional bits requirement similar.

A reconstruction to the original system configuration can be verified in MATLAB via

```
num2 = conv(b1,a2) + conv(b2,a1)
den2 = conv(a1,a2)
num2 = den2.*D + num2
```

Finally, Table 4.5 compares the different design options for the all-pass designs. This time it seems that the LWDF filter is best in terms of bit width and as good as others in terms of resources and delay. Operation counts for direct and cascade designs are similar to the three-port design, while the delay path is best for direct, cascade, and two-port LWDF designs. However, since direct or cascade designs need over twice the bit width the overall winner seemed to be the LWDF. We therefore implement the LWDF in the next example.

Example 4.9: All-Pass Three-Port Wave Digital Filter HDL Design

The following VHDL code⁷ shows a possible implementation of this fifth order IIR filter using the lattice wave digital filter architecture from Fig. 4.56, p. 292:

```
-- Description: 5th order Lattice Wave Digital Filter
-- Coefficients gamma:
```

⁷ The equivalent Verilog code `iir5lwdf.v` for this example can be found in Appendix A on page 828. Synthesis results are shown in Appendix B on page 881.

```

-- 0.988727 -0.000528 -1.995400 -0.000282 -1.985024

LIBRARY ieee;
USE ieee.std_logic_1164.ALL; USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

LIBRARY ieee_proposed;
use ieee_proposed.fixed_float_types.all;
use ieee_proposed.fixed_pkg.all;
use ieee_proposed.float_pkg.all;
-----
ENTITY iir5lwdf IS                                     ----> Interface
PORT (clk : IN STD_LOGIC; -- System clock
      reset : IN STD_LOGIC; -- System reset
      x_in : IN STD_LOGIC_VECTOR(31 DOWNTO 0); -- System input
      y_ap1out : OUT STD_LOGIC_VECTOR(31 DOWNTO 0); -- AP1 out
      y_ap2out : OUT STD_LOGIC_VECTOR(31 DOWNTO 0); -- AP2 out
      y_ap3out: OUT STD_LOGIC_VECTOR(31 DOWNTO 0); -- AP3 out
      y_out : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)); -- System
                                                -- output
END;
-----
ARCHITECTURE fpga OF iir5lwdf IS
-- Coefficients gamma
CONSTANT g1 : SFIXED(6 DOWNTO -15) :=
          TO_SFIXED(0.988727, 6,-15);
CONSTANT g2 : SFIXED(6 DOWNTO -15) :=
          TO_SFIXED(-0.000528, 6,-15);
CONSTANT g3 : SFIXED(6 DOWNTO -15) :=
          TO_SFIXED(-1.995400, 6,-15);
CONSTANT g4 : SFIXED(6 DOWNTO -15) :=
          TO_SFIXED(-0.000282, 6,-15);
CONSTANT g5 : SFIXED(6 DOWNTO -15) :=
          TO_SFIXED(-1.985024, 6,-15);
-- Internal signals
SIGNAL c1, c2, c3, l2, l3 : SFIXED(6 DOWNTO -15) :=
          (OTHERS => '0');
SIGNAL x, ap1, ap2, ap3, ap3r, y : SFIXED(6 DOWNTO -15)
          := (OTHERS => '0');
SIGNAL x32, y_sfix, y_ap1, y_ap2, y_ap3 :
          SFIXED(15 DOWNTO -16);
BEGIN
  x32 <= TO_SFIXED(x_in, x32); -- Redefine bits as FIX 16.16
  x <= resize(x32, x); -- Internal precision is 6.15 format

  P1: PROCESS (clk, x, reset)      ----> Behavioral Style
    VARIABLE p1, a4, a5, a6, a8, a9, a10 :
          SFIXED(6 DOWNTO -15) := (OTHERS => '0'); -- No FFs
  BEGIN -- First equations without inferring registers
    IF reset = '1' THEN -- Reset all register
      y <= (OTHERS => '0');
      c1 <= (OTHERS => '0'); ap1 <= (OTHERS => '0');
      c2 <= (OTHERS => '0'); l2 <= (OTHERS => '0');

```

```

        ap2 <= (OTHERS => '0');
        c3 <= (OTHERS => '0'); l3 <= (OTHERS => '0');
        ap3 <= (OTHERS => '0'); ap3r <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN -- AP LWDF form
    -- 1. AP section is 1. order
        p1 := resize(g1 * (c1-x),x,fixed_wrap,fixed_truncate);
        c1 <= resize(x + p1,x,fixed_wrap,fixed_truncate);
        ap1 <= resize(c1 + p1,x,fixed_wrap,fixed_truncate);
    -- 2. AP section is 2. order
        a4 := resize(ap1-l2+ c2,x,fixed_wrap,fixed_truncate);
        a5 := resize(a4 * g2+c2,x,fixed_wrap,fixed_truncate);
        a6 := resize(a4 * g3-12,x,fixed_wrap,fixed_truncate);
        c2 <= resize(a5,x,fixed_wrap,fixed_truncate);
        l2 <= resize(a6,x,fixed_wrap,fixed_truncate);
        ap2 <= resize(-a5-a6-a4,x);
    -- 3. AP section is 2. order
        a8 := resize(x - l3 +c3,x,fixed_wrap,fixed_truncate);
        a9 := resize(a8 * g4+c3,x,fixed_wrap,fixed_truncate);
        a10 := resize(a8 *g5-13,x,fixed_wrap,fixed_truncate);
        c3 <= resize(a9,x,fixed_wrap,fixed_truncate);
        l3 <= resize(a10,x,fixed_wrap,fixed_truncate);
        ap3 <=resize(-a9-a10-a8,x,fixed_wrap,fixed_truncate);
        ap3r <= ap3; -- extra register due to AP1
    -- Output adder
        y <= resize(ap3r + ap2,x,fixed_wrap,fixed_truncate);
    END IF;                                     -- Output sum
END PROCESS;

-- Convert to 16.16 sfixed number
y_sfix <= resize(y, y_sfix,fixed_wrap,fixed_truncate);
y_ap1 <= resize(ap1, y_sfix,fixed_wrap,fixed_truncate);
y_ap2 <= resize(ap2, y_sfix,fixed_wrap,fixed_truncate);
y_ap3 <= resize(ap3, y_sfix,fixed_wrap,fixed_truncate);
-- Redefine bits as 32 bit SLV
y_out <= to_slv(y_sfix);
y_ap1out <= to_slv(y_ap1);
y_ap2out <= to_slv(y_ap2);
y_ap3out <= to_slv(y_ap3);

END fpga;

```

The design ENTITY includes, besides the system input and outputs, the output of the three all-pass wave digital filter sections. As shown in Fig. 4.56 the upper part consists of one first order and one second order section and the lower all-pass of one second order section. The five γ CONSTANT coefficients are conveniently defined using the TO_SFIXED function. A single PROCESS for all three all-pass sections follows. We first include an asynchronous reset to all registers. After the **rising_edge** statement the assignments for the WDF follows. Each assignment will also infer a register. We apply the **resize** function after each arithmetic operation to avoid overflow or bit growth. The parameter **fixed_wrap** and **fixed_truncate** are used to avoid the default thresholding. Without the **fixed_wrap** more than twice as many LEs would be needed. Finally we convert the section signals and the filter output first to **sfixed** and then to SLV data type. The design uses 764 LEs, 12 embedded

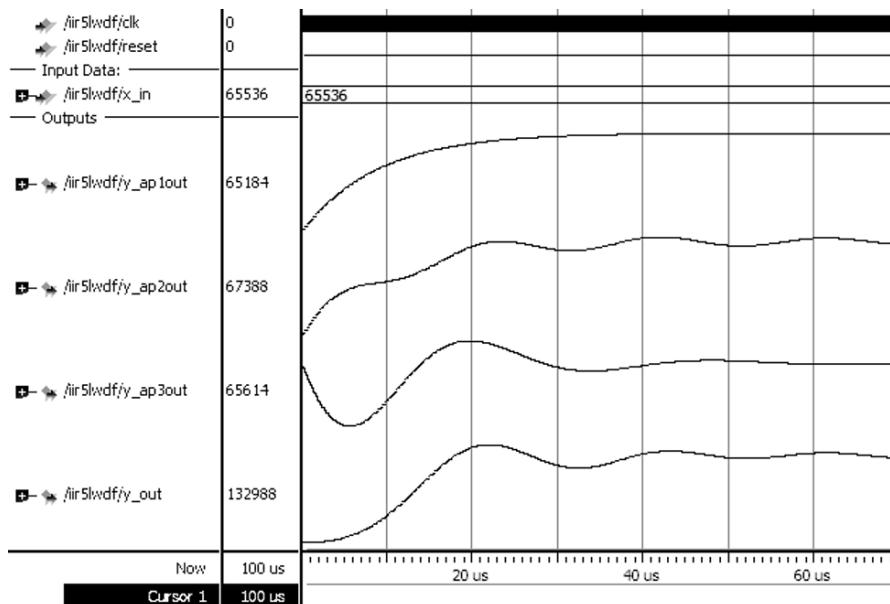


Fig. 4.59. Step response for MODELSIM simulation of the fifth order all-pass wave digital lattice filter

multipliers, and has an $F_{max} = 55.97$ MHz registered performance using the TimeQuest slow 85C model. The response of the filter to a step function with amplitude $1.0 = 65536_{10}$ for 16.16 `sfixed` format, shown in Fig. 4.59, agrees with the MATLAB simulated results presented in Fig. 4.23b, p. 253. The 60 μ s simulation requires 600 clock cycles for a 100-ns system clock. The step response maximum occurs after about 215 clock cycles.

4.9

Compared to the best non-all-pass design from Example 4.7 (p. 255) the LWDF design shows an improvement in the use embedded multipliers (50% less). LEs and speed are a little better with the modified parallel BiQuad design. Table 4.6 gives an overview of the synthesis results of the narrow band filter designs discussed in this chapter.

Exercises

Note: If you have no prior experience with the Quartus II software, refer to the case study found in Sect. 1.4.3, p. 32. If not otherwise noted use the EP4CE115F29C7 from the Cyclone IV E family for the Quartus II synthesis evaluations.

Table 4.6. HDL synthesis result for narrow band IIR filters

Architecture	Wrap	LEs	Mult. 9×9	F _{max} (MHz)
Direct	—	6279	128	29.96
Direct	✓	2474	128	46.99
Parallel BiQuad	—	1871	51	54.00
Parallel BiQuad	✓	624	51	87.69
All-pass three-port LWDF	—	1465	12	33.83
All-pass three-port LWDF	✓	764	12	55.97

4.1: A filter has the following specification: sampling frequency 2 kHz; passband 0–0.4 kHz, stopband 0.5–1 kHz; passband ripple 3 dB, and stopband ripple 48 dB. Use the MATLAB software and the “Interactive Lowpass Filter Design” demo or the `fdatool` from the Signal Processing Toolbox for the filter design to:

- (a1) Design a Butterworth filter (called BUTTER).
- (a2) Determine the filter length and the absolute ripple in the passband.
- (b1) Design a Chebyshev type I filter (called CHEBY1).
- (b2) Determine the filter length and the absolute ripple in the passband.
- (c1) Design a Chebyshev type II filter (called CHEBY2).
- (c2) Determine the filter length and the absolute ripple in the passband.
- (d1) Design an elliptic filter (called ELLIP).
- (d2) Determine the filter length and the absolute ripple in the passband.

4.2: (a) Compute the maximum bit growth for a first order IIR filter with a pole at $z_\infty = 3/4$.

(a2) Use the MATLAB or C software to verify the bit growth using a step response of the first order IIR filter with a pole at $z_\infty = 3/4$.

(b) Compute the maximum bit growth for a first order IIR filter with a pole at $z_\infty = 3/8$.

(b2) Use the MATLAB or C software to verify the bit growth using a step response of the first order IIR filter with a pole at $z_\infty = 3/8$.

(c) Compute the maximum bit growth for a first order IIR filter with a pole at $z_\infty = p$.

4.3: (a) Implement a first order IIR filter with a pole at $z_{\infty 0} = 3/8$ and 12-bit input width, using Quartus II.

(b) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks).

(c) Simulate the design with an input impulse of amplitude 100.

(d) Compute the maximum bit growth for the filter.

(e) Verify the result from (d) with a simulation of the step response with amplitude 100.

4.4: (a) Implement a first order IIR filter with a pole at $z_{\infty 0} = 3/8$, 12-bit input width, and a look-ahead of one step, using Quartus II.

(b) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks).

(c) Simulate the design with an input impulse of amplitude 100.

4.5: (a) Implement a first order IIR filter with a pole at $z_{\infty 0} = 3/8$, 12-bit input width, and a parallel design with two paths, using Quartus II.

- (b) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks).
 (c) Simulate the design with an input impulse of amplitude 100.

4.6: (a) Implement a first order IIR filter as in Example 4.1 (p. 226), using a 15-bit `std_logic_vector`, and implement the adder with two `lpm_add_sub` megafunctions, using Quartus II.

- (b) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks).

- (c) Simulate the design with an input impulse of amplitude 1000, and compare the results to Fig. 4.3 (p. 228).

4.7: (a) Implement a first order pipelined IIR filter from Example 4.3 (p. 241) using a 15-bit `std_logic_vector`, and implement the adder with four `lpm_add_sub` megafunctions, using Quartus II.

- (b) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks).

- (c) Simulate the design with an input impulse of amplitude 1000, and compare the results to Fig. 4.15 (p. 243).

4.8: Shajaan and Sorensen have shown that an IIR Butterworth filter can be efficiently designed by implementing the coefficients as signed-power-of-two (SPT) values [122]. The transfer function of a cascade filter with N sections

$$F(z) = \prod_{l=1}^N S[l] \frac{b[l, 0] + b[l, 1]z^{-1} + b[l, 2]z^{-2}}{a[l, 0] + a[l, 1]z^{-1} + a[l, 2]z^{-2}} \quad (4.52)$$

should be implemented using the second order sections shown in Fig. 4.11 (p. 236). A tenth order filter, as discussed in Example 4.2 (p. 237), can be realized with the following SPT filter coefficients [122]:

l	$S[l]$	$1/a[l, 0]$	$a[l, 1]$	$a[l, 2]$
1	2^{-1}	1	$-1 - 2^{-4}$	$1 - 2^{-2}$
2	2^{-1}	2^{-1}	$-1 - 2^{-1}$	$1 - 2^{-5}$
3	2^{-1}	2^{-1}	$-1 - 2^{-1}$	$2^{-1} + 2^{-5}$
4	1	2^{-1}	$-1 - 2^{-2}$	$2^{-2} + 2^{-5}$
5	2^{-1}	2^{-1}	$-1 - 2^{-1}$	$2^{-2} + 2^{-4}$

We choose $b[0] = b[2] = 0.5$ and $b[1] = 1$ because the zeros of the Butterworth filter are all at $z = -1$.

- (a) Compute and plot the transfer function of the first BiQuad and the complete filter.

- (b) Implement and simulate the first BiQuad for 8-bit inputs.

- (c) Build and simulate the five-stage filter with Quartus II.

- (d) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) of the filter.

4.9: (a) Design a tenth order order lowpass Butterworth filter using MATLAB with a cut-off frequency of 0.3 the sampling frequency, i.e., `[b,a]=butter(10,0.3)`.

- (b) Plot the transfer function using `freqz()` for ∞ bits (i.e., real coefficients) and 12-bit fractional bits. What is the stopband suppression in dB at 0.5 of the Nyquist frequency?

Hint: `round(a*2^B)/2^B` has B fractional bits.

- (c) Generate a pole/zero plot using `zplane()` for ∞ bits and 12 fractional bits.

- (d) Plot the impulse response for an impulse of amplitude 100 using `filter()` and `stem()` for coefficients with 12-bit fractional bits. Also plot the response to impulse of amplitude 100 of the recursive part only, i.e., set the FIR part to `b=[1]`;
 (e) For the 12 fractional bit filter determine using the `csd3e.exe` program from the CD the CSD representation for all coefficients `a` and `b` from part (a).

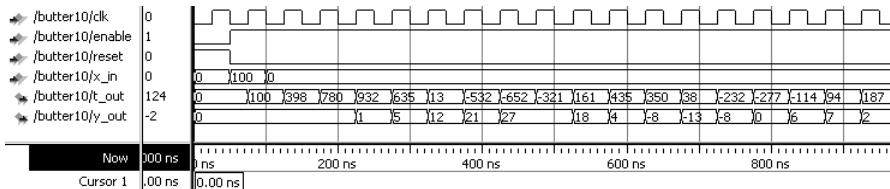


Fig. 4.60. IIR Butterworth test bench for Exercise 4.10

4.10: (a) Using the results from Exercise 4.9 develop the VHDL code for the tenth order Butterworth filter for 8-bit inputs. As internal data format use a 14.12 bit format, i.e., 14 integer and 12 fractional bits. You need to scale the input and output by 2^{12} and use an internal 26-bit format. Use the direct form II, i.e., Fig. 4.8 (p. 234) for your design. Make sure that the transfer function of Fig. 4.8 and the MATLAB representation of the transfer functions match.

Recommendation: you can start with the recursive part first and try to match the simulation from Exercise 4.9(d). Then add the nonrecursive part.

- (b) Add an active-high enable and active-high asynchronous reset to the design.
 (c) Try to match the simulation from Exercise 4.9(d) shown in simulation Fig. 4.60, where `t_out` is the output of the recursive part.
 (d) For the device EP4CE115F29C7 from the Cyclone IV E family determine the resources (LEs, multipliers, and M9Ks) and registered performance `Fmax` using the TimeQuest slow 85C model.

4.11: (a) Design the PREP benchmark 5 shown in Fig. 4.61a with the Quartus II software. The design has a 4×4 unsigned array multiplier followed by an 8-bit accumulator. If `mac = '1'` accumulation is performed otherwise the adder output `s` shows the multiplier output without adding `q`. `rst` is an asynchronous reset and the 8-bit register is positive-edge triggered via `clk`, see the simulation in Fig. 4.61c for the function test.

(b) Determine the registered performance `Fmax` using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) for a single copy. Compile the HDL file with the synthesis Optimization Technique set to Speed, Balanced or Area; this can be found in the Analysis & Synthesis Settings section under EDA Tool Settings in the Assignments menu. Which synthesis options are optimal for size or registered performance `Fmax` using the TimeQuest slow 85C model?

Select one of the following devices:

- (b1) EP4CE115F29C7 from the Cyclone IV E family
 (b2) EP2C35F672C6 from the Cyclone II family
 (b3) EPM7128SLC84-7 from the MAX7000S family
 (c) Design the multiple instantiation for benchmark 5 as shown in Fig. 4.61b.
 (d) Determine the registered performance `Fmax` using the TimeQuest slow 85C

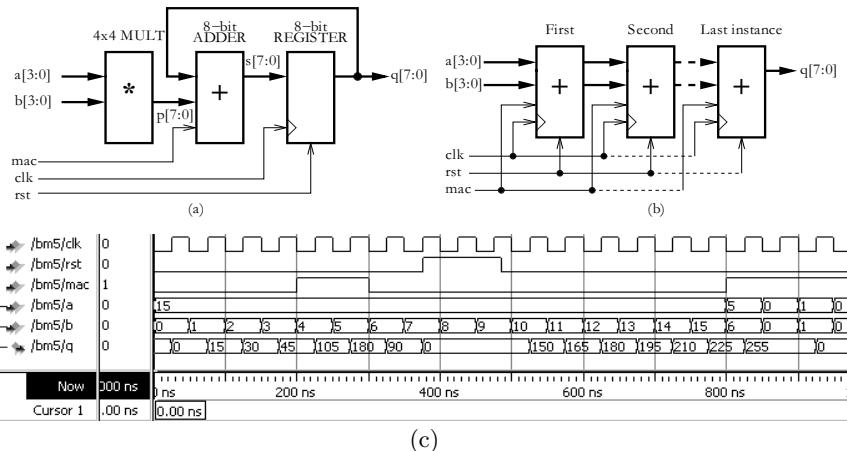


Fig. 4.61. PREP benchmark 5. (a) Single design. (b) Multiple instantiation. (c) Test bench to check function

model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for the design with the maximum number of instantiations of PREP benchmark 5. Use the optimal synthesis option you found in (b) for the following devices:

- (d1) EP4CE115F29C7 from the Cyclone IV E family
- (d2) EP2C35F672C6 from the Cyclone II family
- (d3) EPM7128SLC84-7 from the MAX7000S family

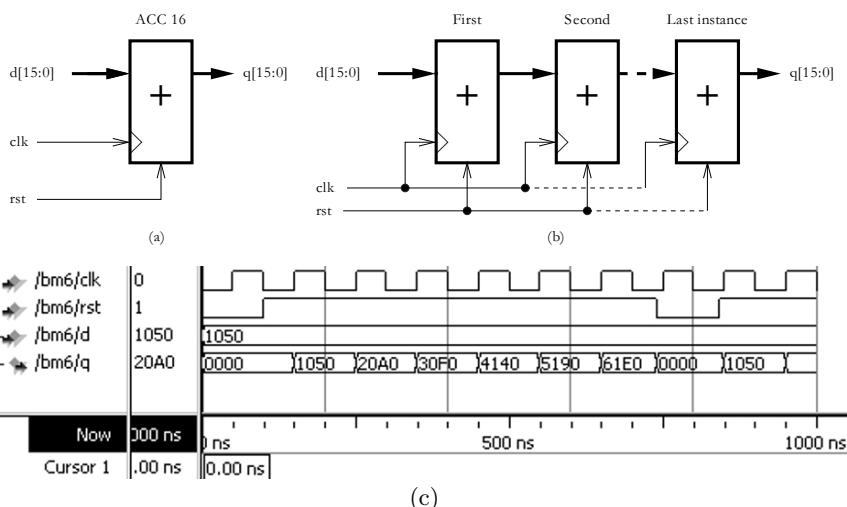


Fig. 4.62. PREP benchmark 6. (a) Single design. (b) Multiple instantiation. (c) Test bench to check function

4.12: (a) Design the PREP benchmark 6 shown in Fig. 4.62a with the Quartus II software. The design is positive-edge triggered via `clk` and includes a 16-bit accumulator with an asynchronous reset `rst`; see the simulation in Fig. 4.62c for the function test.

(b) Determine the registered performance `Fmax` using the `TimeQuest` slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for a single copy. Compile the HDL file with the synthesis `Optimization Technique` set to `Speed`, `Balanced` or `Area`; this can be found in the `Analysis & Synthesis Settings` section under `EDA Tool Settings` in the `Assignments` menu. Which synthesis options are optimal for size or registered performance `Fmax` using the `TimeQuest` slow 85C model?

Select one of the following devices:

- (b1) EP4CE115F29C7 from the Cyclone IV E family
 - (b2) EP2C35F672C6 from the Cyclone II family
 - (b3) EPM7128SLC84-7 from the MAX7000S family
- (c) Design the multiple instantiation for benchmark 6 as shown in Fig. 4.62b.
- (d) Determine the registered performance `Fmax` using the `TimeQuest` slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for the design with the maximum number of instantiations of PREP benchmark 6. Use the optimal synthesis option you found in (b) for the following devices:
- (d1) EP4CE115F29C7 from the Cyclone IV E family
 - (d2) EP2C35F672C6 from the Cyclone II family
 - (d3) EPM7128SLC84-7 from the MAX7000S family

5. Multirate Signal Processing

Introduction

A frequent task in digital signal processing is to adjust the sampling rate according to the signal of interest. Systems with different sampling rates are referred to as *multirate* systems. In this chapter, two typical examples will illustrate decimation and interpolation in multirate DSP systems. We will then introduce polyphase notation, and will discuss some efficient decimator designs. At the end of the chapter we will discuss filter banks and a quite new, highly celebrated addition to the DSP toolbox: wavelet analysis.

5.1 Decimation and Interpolation

If, after A/D conversion, the signal of interest can be found in a small frequency band (typically, lowpass or bandpass), then it is reasonable to filter with a lowpass or bandpass filter and to reduce the sampling rate. A narrow filter followed by a downampler is usually referred to as a *decimator* [88].¹ The filtering, downsampling, and the effect on the spectrum is illustrated in Fig. 5.1.

We can reduce the sampling rate up to the limit called the “Nyquist rate,” which says that the sampling rate must be higher than the bandwidth of the signal, in order to avoid aliasing. Aliasing is demonstrated in Fig. 5.2 for a lowpass signal. Aliasing is irreparable, and should be avoided at all cost.

For a bandpass signal, the frequency band of interest must fall within an *integer band*. If f_s is the sampling rate, and R is the desired downsampling factor, then the band of interest must fall between

$$k \frac{f_s}{2R} < f < (k + 1) \frac{f_s}{2R} \quad k \in \mathbb{N}. \quad (5.1)$$

If it does not, there may be aliasing due to “copies” from the negative frequency bands, although the sampling rate may still be higher than the Nyquist rate, as shown in Fig. 5.3.

Increasing the sampling rate can be useful, in the D/A conversion process, for example. Typically, D/A converters use a sample-and-hold of first

¹ Some authors refer to a downampler as a decimator.

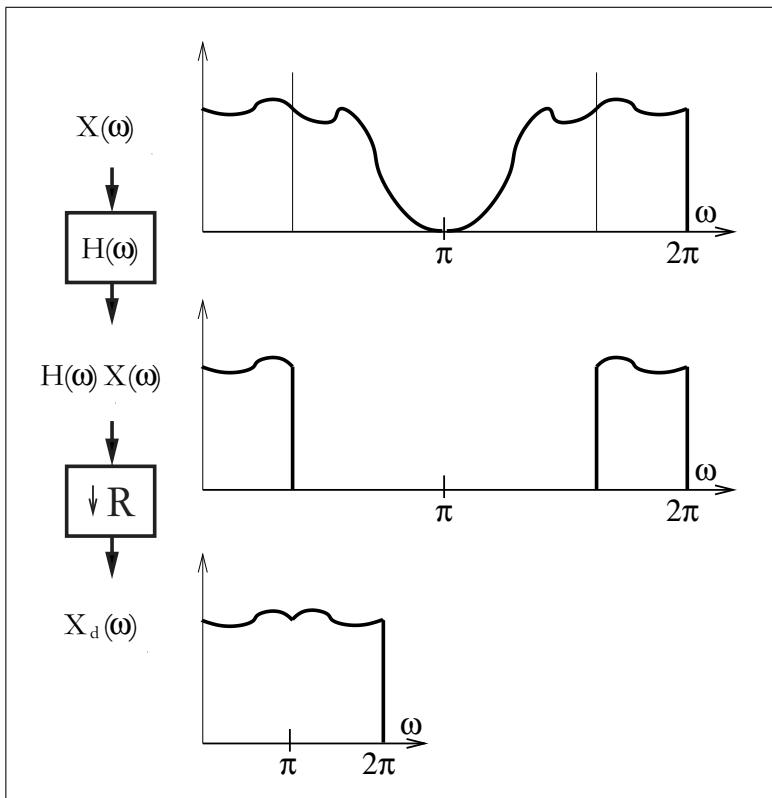


Fig. 5.1. Decimation of signal $x[n]$ $\circ-\bullet X(\omega)$

order at the output, which produces a step-like output function. This can be compensated for with an analog $1/\text{sinc}(x)$ compensation filter, but most often a digital solution is more efficient. We can use, in the digital domain, an *expander* and an additional filter to get the desired frequency band. We note, from Fig. 5.4, that the introduced zeros produce an extra copy of the baseband spectrum that must first be removed before the signal can be processed with the D/A converter. The much smoother output signal of such an *interpolation*² can be seen in Fig. 5.5b.

5.1.1 Noble Identities

When manipulating signal flow graphs of multirate systems it is sometimes useful to rearrange the filter and downampler/expander, as shown in Fig. 5.6. These are the so-called “Noble” relations [123]. For the decimator, it follows:

$$(\downarrow R) F(z) = F(z^R) (\downarrow R), \quad (5.2)$$

² Some authors refer to the expander as an interpolator.

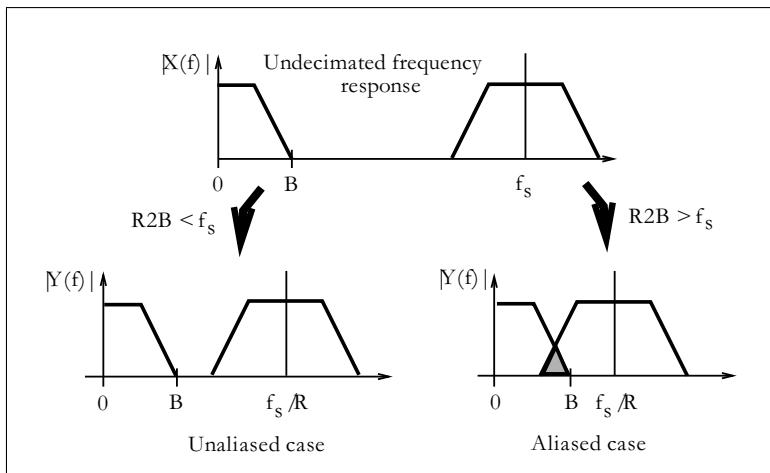


Fig. 5.2. Unaliased and aliased decimation cases

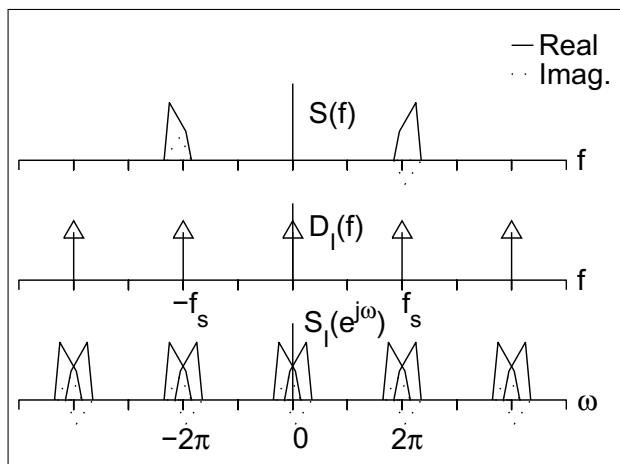


Fig. 5.3. Integer band violation. (© VDI Press [4])

i.e., if the downsampling is done first, we can reduce the filter length $F(z^R)$ by a factor of R .

For the interpolator, the Noble relation is defined as

$$F(z) (\uparrow R) = (\uparrow R) F(z^R), \quad (5.3)$$

i.e., in an interpolation putting the filter before the expander results in an R -times shorter filter.

These two identities will become very useful when we discuss polyphase implementation in Sect. 5.2 (p. 309).

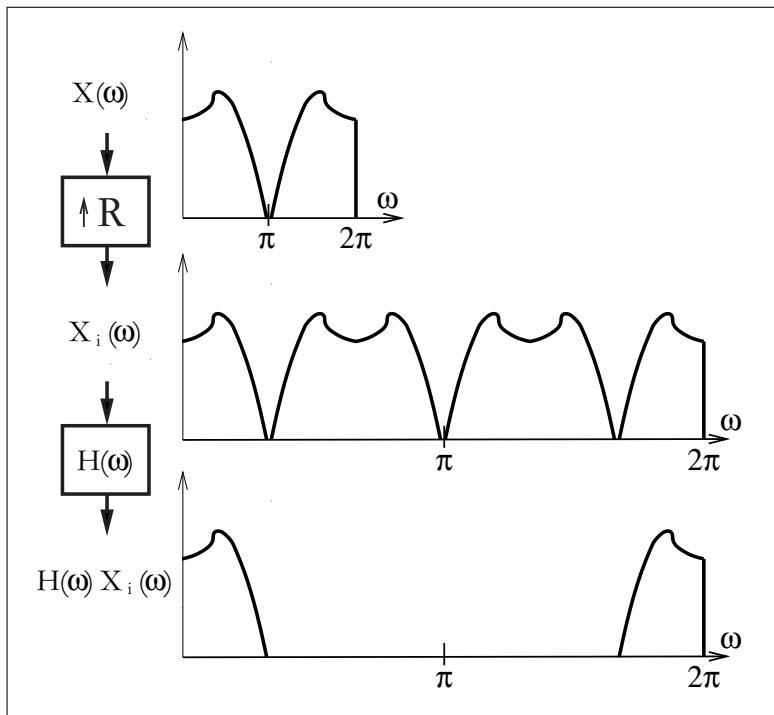


Fig. 5.4. Interpolation example. $R = 3$ for $x[n] \circ \bullet X(\omega)$

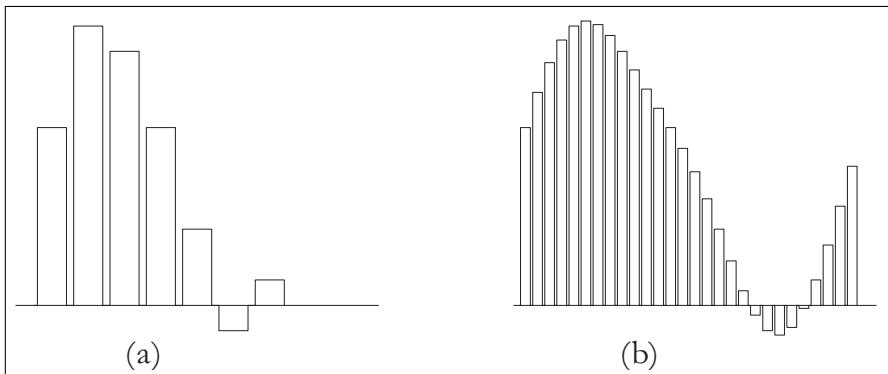


Fig. 5.5. D/A conversion. (a) Low oversampling, high degradation. (b) High oversampling, low degradation

5.1.2 Sampling Rate Conversion by Rational Factor

If the input and output rate of a multirate system is *not* an integer factor, then a rational change factor R_1/R_2 in the sampling rate can be used. More

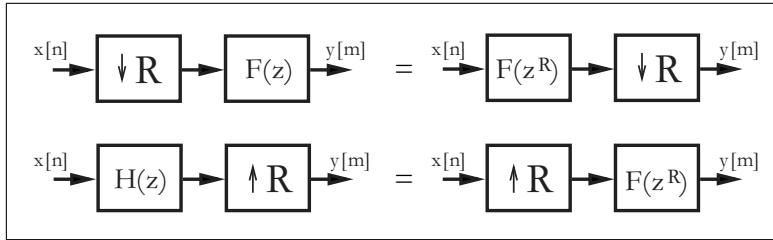


Fig. 5.6. Equivalent multirate systems (Noble relation)

precisely, we first use an interpolator to increase the sampling rate by R_1 , and then use a decimator to downsample by R_2 . Since the filters used for interpolation and decimation are both lowpass filters, it follows, from the upper configuration in Fig. 5.7, that we only need to implement the lowpass filter with the smaller passband frequency, i.e.,

$$f_p = \min\left(\frac{\pi}{R_1}, \frac{\pi}{R_2}\right). \quad (5.4)$$

This is graphically interpreted in the lower configuration of Fig. 5.7. We will discuss later in Sect. 5.6, p. 345 different design options of this system.

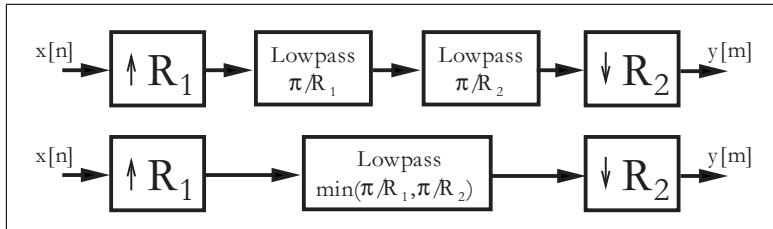


Fig. 5.7. Noninteger decimation system. (*upper*) Cascade of an interpolator and a decimator. (*lower*) Result combining the lowpass filters

5.2 Polyphase Decomposition

Polyphase decomposition is very useful when implementing decimation or interpolation in IIR or FIR filter and filter banks. To illustrate this, consider the polyphase decomposition of an FIR decimation filter. If we add downsampling by a factor of R to the FIR filter structure shown in Fig. 3.1 (p. 180), we find that we only need to compute the outputs $y[n]$ at time instances

$$y[0], y[R], y[2R], \dots . \quad (5.5)$$

It follows that we do not need to compute all sums-of-product $f[k]x[n - k]$ of the convolution. For instance, $x[0]$ only needs to be multiplied by

$$f[0], f[R], f[2R], \dots . \quad (5.6)$$

Besides $x[0]$, these coefficients only need to be multiplied by

$$x[R], x[2R], \dots . \quad (5.7)$$

It is therefore reasonable to split the input signal first into R separate sequences according to

$$\begin{aligned} x[n] &= \sum_{r=0}^{R-1} x_r[n] \\ x_0[n] &= \{x[0], x[R], \dots\} \\ x_1[n] &= \{x[1], x[R+1], \dots\} \\ &\vdots \\ x_{R-1}[n] &= \{x[R-1], x[2R-1], \dots\} \end{aligned}$$

and also to split the filter $f[n]$ into R sequences

$$\begin{aligned} f[n] &= \sum_{r=0}^{R-1} f_r[n] \\ f_0[n] &= \{f[0], f[R], \dots\} \\ f_1[n] &= \{f[1], f[R+1], \dots\} \\ &\vdots \\ f_{R-1}[n] &= \{f[R-1], f[2R-1], \dots\}. \end{aligned}$$

Figure 5.8 shows a decimator filter implemented using polyphase decomposition. Such a decimator can run R times faster than the usual FIR filter followed by a downsampler. The filters $f_r[n]$ are called polyphase filters, because they all have the same magnitude transfer function, but they are separated by a sample delay, which introduces a phase offset.

A final example illustrates the polyphase decomposition.

Example 5.1: Polyphase Decimator Filter

Consider a Daubechies length-4 filter with $G(z)$ and $R = 2$.

$$\begin{aligned} G(z) &= ((1 + \sqrt{3}) + (3 + \sqrt{3})z^{-1} + (3 - \sqrt{3})z^{-2} + (1 - \sqrt{3})z^{-3}) \frac{1}{4\sqrt{2}} \\ G(z) &= 0.48301 + 0.8365z^{-1} + 0.2241z^{-2} - 0.1294z^{-3}. \end{aligned}$$

Quantizing the filter to 8 bits of precision results in the following model:

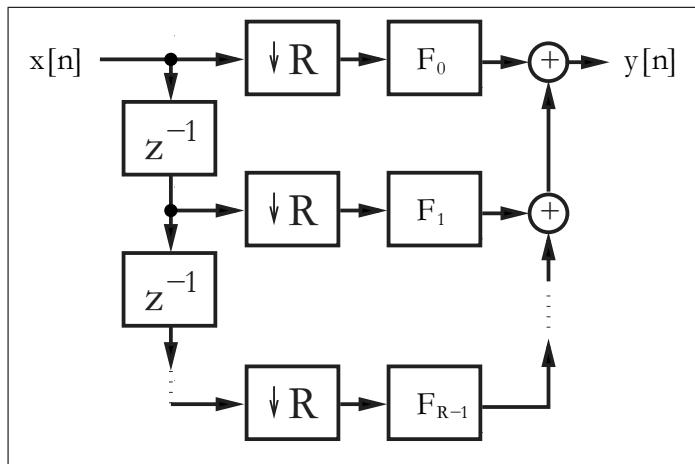


Fig. 5.8. Polyphase realization of a decimation filter

$$G(z) = (124 + 214z^{-1} + 57z^{-2} - 33z^{-3}) / 256$$

$$\begin{aligned} G(z) &= G_0(z^2) + z^{-1}G_1(z^2) \\ &= \underbrace{\left(\frac{124}{256} + \frac{57}{256}z^{-2}\right)}_{G_0(z^2)} + z^{-1} \underbrace{\left(\frac{214}{256} - \frac{33}{256}z^{-2}\right)}_{G_1(z^2)}, \end{aligned}$$

and it follows that:

$$G_0(z) = \frac{124}{256} + \frac{57}{256}z^{-1} \quad G_1(z) = \frac{214}{256} - \frac{33}{256}z^{-1}. \quad (5.8)$$

The following VHDL code³ shows the polyphase implementation for DB4:

```

PACKAGE n_bits_int IS           -- User defined types
  SUBTYPE S8 IS INTEGER RANGE -128 TO 127;
  SUBTYPE S9 IS INTEGER RANGE -2**8 TO 2**8-1;
  SUBTYPE S17 IS INTEGER RANGE -2**16 TO 2**16-1;
  TYPE A0_3S17 IS ARRAY (0 TO 3) of S17;
END n_bits_int;

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-----
ENTITY db4poly IS           -----> Interface
  PORT (clk      : IN STD_LOGIC; -- System clock
        reset    : IN STD_LOGIC; -- Asynchronous reset
        x_in     : IN S8;       -- System input

```

³ The equivalent Verilog code `db4poly.v` for this example can be found in Appendix A on page 829. Synthesis results are shown in Appendix B on page 881.

```

clk2      : OUT STD_LOGIC; -- Clock divided by 2
x_e, x_o : OUT S17;       -- Even/odd x_in
g0, g1   : OUT S17;       -- Poly filter 0/1
y_out    : OUT S9);       -- System output
END db4poly;
-----
ARCHITECTURE fpga OF db4poly IS

TYPE STATE_TYPE IS (even, odd);
SIGNAL state           : STATE_TYPE;
SIGNAL x_odd, x_even, x_wait : S8 := 0;
SIGNAL clk_div2        : STD_LOGIC;
-- Arrays for multiplier and taps:
SIGNAL r   : A0_3S17 := (0,0,0,0);
SIGNAL x33, x99, x107   : S17 := 0;
SIGNAL y     : S17;

BEGIN

Multiplex: PROCESS(reset, clk) ----> Split into even and
BEGIN                                     -- odd samples at clk rate
  IF reset = '1' THEN                  -- Asynchronous reset
    state <= even;
    clk_div2 <= '0'; x_even <= 0; x_odd <= 0; x_wait <= 0;
  ELSIF rising_edge(clk) THEN
    CASE state IS
      WHEN even =>
        x_even <= x_in;
        x_odd  <= x_wait;
        clk_div2 <= '1';
        state <= odd;
      WHEN odd =>
        x_wait <= x_in;
        clk_div2 <= '0';
        state <= even;
    END CASE;
  END IF;
END PROCESS Multiplex;

AddPolyphase: PROCESS (reset, clk_div2, x_odd, x_even,
                      x33, x99, x107)
VARIABLE m  : A0_3S17;
BEGIN
-- Compute auxiliary multiplications of the filter
  x33  <= x_odd * 32 + x_odd;
  x99  <= x33 * 2 + x33;
  x107 <= x99 + 8 * x_odd;
-- Compute all coefficients for the transposed filter
  m(0) := 4 * (32 * x_even - x_even);          -- m[0] = 127
  m(1) := 2 * x107;                            -- m[1] = 214
  m(2) := 8 * (8 * x_even - x_even) + x_even; -- m[2] = 57
  m(3) := x33;                                 -- m[3] = -33

```

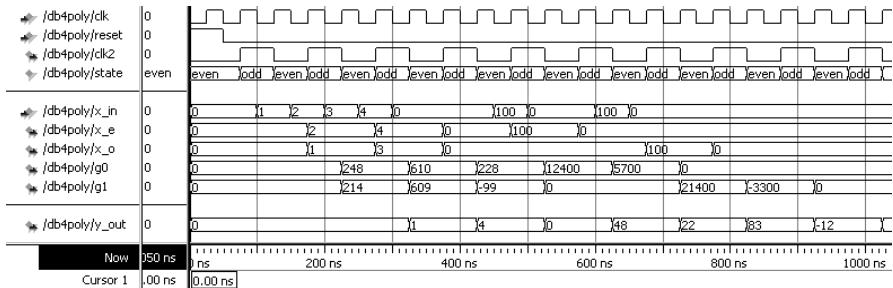


Fig. 5.9. VHDL simulation of the polyphase implementation of the length-4 Daubechies filter

```

        IF reset = '1' THEN -- Asynchronous clear all registers
          FOR k IN 0 TO 3 LOOP
            r(k) <= 0;
          END LOOP;
          y <= 0;
        -----> Compute the filters and infer registers
        ELSIF falling_edge(clk_div2) THEN
        ----- Compute filter G0
          r(0) <= r(2) + m(0);    -- g[0] = 127
          r(2) <= m(2);           -- g[2] = 57
        ----- Compute filter G1
          r(1) <= -r(3) + m(1);   -- g[1] = 214
          r(3) <= m(3);           -- g[3] = -33
        ----- Add the polyphase components
          y <= r(0) + r(1);
        END IF;
      END PROCESS AddPolyphase;

      x_e <= x_even; -- Provide some test signal as outputs
      x_o <= x_odd;
      clk2 <= clk_div2;
      g0 <= r(0);
      g1 <= r(1);

      y_out <= y / 256; -- Connect to output

    END fpga;
  
```

The first PROCESS is the FSM, which includes the control flow and the splitting of the input stream at the sampling rate into even and odd samples. The second PROCESS includes the reduced adder graph (RAG) multiplier, and the last PROCESS hosts the two filters in a transposed structure. Although the output is scaled, there is potential growth by the amount $\sum |g_k| = 1.673 < 2^1$. Therefore the output y_{out} was chosen to have an additional guard bit. The design uses 167 LEs, no embedded multiplier, and has an $F_{max}=618.43$ MHz registered performance using the TimeQuest slow 85C model.

A simulation of the filter is shown in Fig. 5.9. The first four input samples are a triangle function to demonstrate the splitting into even and odd samples.

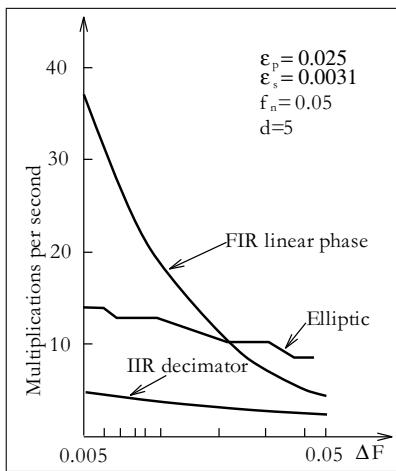


Fig. 5.10. Comparison of computational effort for decimators $\Delta F = f_p - f_s$. (Sampling frequency $2 \times f_n = 0.1$)

Impulses with an amplitude of 100 are used to verify the coefficients of the two polyphase filters. Note that the filter is *not* shift invariant. 5.1

From the VHDL simulation shown in Fig. 5.9, it can be seen that such a decimator is no longer shift invariant, resulting in a technically nonlinear system. This can be validated by applying a single impulse. Initializing at an even-indexed sample, the response is $G_0(z)$, while for an odd-indexed sample, the response is $G_1(z)$.

5.2.1 Recursive IIR Decimator

It is also possible to apply polyphase decomposition to recursive filters and to get the speed benefit, if we follow the idea from Martinez and Parks [110], in the transfer function

$$F(z) = \frac{\sum_{l=0}^{L-1} a[l]z^{-l}}{1 - \sum_{l=1}^{K-1} b[l]z^{-lR}}. \quad (5.9)$$

i.e., the recursive part has only each R^{th} coefficient. We have already discussed such a design in the context of IIR filters (Fig. 4.17, p. 246). Figure 5.10 shows that, depending on the transition width ΔF of the filter, an IIR decimator offers substantial savings compared with an FIR decimator.

5.2.2 Fast-running FIR Filter

An interesting application of polyphase decomposition is the so-called *fast-running* FIR filter. The basic idea of this filter is the following: If we decompose the input signal $x[n]$ into R polyphase components, we can use Winograd's short convolution algorithms to implement a fast filter. Let us demonstrate this with an example for $R = 2$.

Example 5.2: Fast-Running FIR filter

We decompose the input signal $X(z)$ and filter $F(z)$ into even and odd polyphase components, i.e.,

$$X(z) = \sum_n x[n]z^{-n} = X_0(z^2) + z^{-1}X_1(z^2) \quad (5.10)$$

$$F(z) = \sum_n f[n]z^{-n} = F_0(z^2) + z^{-1}F_1(z^2). \quad (5.11)$$

The convolution in the time domain of $x[n]$ and $f[n]$ yields a polynomial multiply in the z -domain. It follows for the output signal $Y(z)$ that:

$$Y(z) = Y_0(z^2) + z^{-1}Y_1(z^2) \quad (5.12)$$

$$= (X_0(z^2) + z^{-1}X_1(z^2))(F_0(z^2) + z^{-1}F_1(z^2)). \quad (5.13)$$

If we split (5.13) into the polyphase components $Y_0(z)$ and $Y_1(z)$ we get

$$Y_0(z) = X_0(z)F_0(z) + z^{-1}X_1(z)F_1(z) \quad (5.14)$$

$$Y_1(z) = X_1(z)F_0(z) + X_0(z)F_1(z). \quad (5.15)$$

If we now compare (5.13) with a 2×2 linear convolution

$$A(z) \times B(z) = (a[0] + z^{-1}a[1])(b[0] + z^{-1}b[1]) \quad (5.16)$$

$$= a[0]b[0] + z^{-1}(a[0]b[1] + a[1]b[0]) + a[1]b[1]z^{-2}, \quad (5.17)$$

we notice that the factors for z^{-1} are the same, but for $Y_0(z)$ we must compute an extra delay to get the right phase relation. Winograd [124] has compiled a list of short convolution algorithms, and a linear 2×2 convolution can be computed using three multiplications and six adds with

$$\begin{aligned} a[0] &= x[0] - x[1] & a[1] &= x[0] & a[2] &= x[1] - x[0] \\ b[0] &= f[0] - f[1] & b[1] &= f[0] & b[2] &= f[1] - f[0] \\ c[k] &= a[k]b[k] & k &= 0, 1, 2 & & \\ y[0] &= c[1] + c[2] & y[1] &= c[1] - c[0]. & & \end{aligned} \quad (5.18)$$

With the help of this short convolution algorithm, we can now define the fast-running filter as follows:

$$\begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & -1 \\ -1 & 1 & 0 \end{bmatrix} \begin{bmatrix} F_0 & 0 & 0 \\ 0 & F_0 + F_1 & 0 \\ 0 & 0 & F_1 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 1 & 0 \\ 1 & -z^{-1} \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \end{bmatrix}. \quad (5.19)$$

Figure 5.11 shows the graphical interpretation.

5.2

If we compare the direct filter implementation with the fast-running FIR filter we must distinguish between hardware effort and average number of

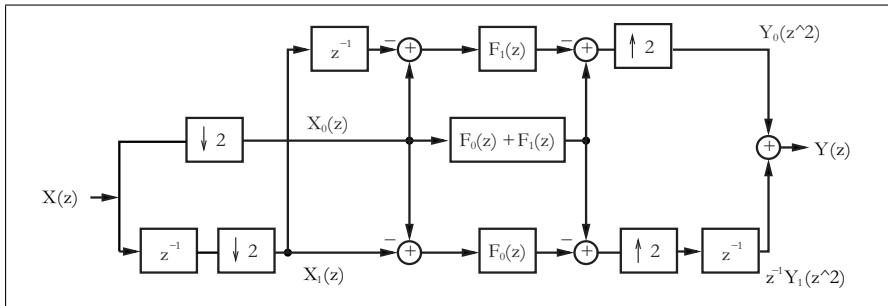


Fig. 5.11. Fast-running FIR filter with $R = 2$

adder and multiplier operations. A direct implementation would have L multipliers and $L - 1$ adders running at full speed. For the fast-running filter we have three filters of length $L/2$ running at half speed. This results in $3L/4$ multiplications per output sample and $(2+2)/2 + 3/2(L/2 - 1) = 3L/4 + 1/2$ additions for the whole filter, i.e., the arithmetic count is about 25% better than in the direct implementation. From an implementation standpoint, we need $3L/2$ multipliers and $4 + 3(L/2 - 1) = 3L/2 + 1$ adders, i.e., the effort is about 50% higher than in the direct implementation. The important feature in Fig. 5.11 is that the fast-running filter basically runs at twice the speed of the direct implementation. Using a higher number R of decomposition may further increase the maximum throughput. The general methodology for R polyphase signals with f_a as input rate is now as follows:

Algorithm 5.3: Fast-Running FIR Filter

- 1) Decompose the input signal into R polyphase signals, using A_e adders to form R sequences at a rate of f_a/R .
- 2) Filter the R sequences with R filters of length L/R .
- 3) Use A_a additions to compute the polyphase representation of the output $Y_k(z)$. Use a final output multiplexer to generate the output signal $Y(z)$.

Note that the computed partial filter of length L/R may again be decomposed, using Algorithm 5.3. Then the question arises: When should we stop the iterative decomposition? Mou and Duhamel [125] have compiled a table with the goal of minimizing the average arithmetic count. Table 5.1 shows the optimal decomposition. The criterion used was a minimum total number of multiplications and additions, which is typical for a MAC-based design. In Table 5.1, all partial filters that should be implemented based on Algorithm 5.3 are underlined.

For a larger length than 60, a fast convolution using the FFT is more efficient, and will be discussed in Chap. 6.

Table 5.1. Computational effort for the recursive FIR decomposition [125, 126]

L	Factors	$M + A$	$\frac{M + A}{L}$	L	Factors	$M + A$	$\frac{M + A}{L}$
2	direct	6	3	22	$\underline{11} \times 2$	668	30.4
3	direct	15	5	24	$\underline{2^2} \times \underline{3} \times 2$	624	26
4	$\underline{2} \times 2$	26	6.5	25	$\underline{5} \times 5$	740	29.6
5	direct	45	9	26	$\underline{13} \times 2$	750	28.9
6	$\underline{3} \times 2$	56	9.33	27	$\underline{3^2} \times 3$	810	30
8	$\underline{2^2} \times 2$	94	11.75	30	$\underline{5} \times \underline{3} \times 2$	912	30.4
9	$\underline{3} \times 3$	120	13.33	32	$\underline{2^4} \times 2$	1006	31.44
10	$\underline{5} \times 2$	152	15.2	33	$\underline{11} \times 3$	1248	37.8
12	$\underline{2} \times \underline{3} \times 2$	192	16	35	$\underline{7} \times 5$	1405	40.1
14	$\underline{7} \times 2$	310	22.1	36	$\underline{2^2} \times \underline{3} \times 3$	1260	35
15	$\underline{5} \times 3$	300	20	39	$\underline{13} \times 3$	1419	36.4
16	$\underline{2^3} \times 2$	314	19.63	55	$\underline{11} \times 5$	2900	52.7
18	$\underline{2} \times \underline{3} \times 3$	396	22	60	$\underline{5} \times \underline{2} \times \underline{3} \times 2$	2784	46.4
20	$\underline{5} \times \underline{2} \times 2$	472	23.6	65	$\underline{13} \times 5$	3345	51.46
21	$\underline{7} \times 3$	591	28.1				

5.3 Hogenauer CIC Filters

A very efficient architecture for a high decimation-rate filter is the “cascade integrator comb” (CIC) filter introduced by Hogenauer [127]. The CIC (also known as the Hogenauer filter), has proven to be an effective element in high-decimation or interpolation systems. One application is in wireless communications, where signals, sampled at RF or IF rates, need to be reduced to baseband. For narrowband applications (e.g., cellular radio), decimation rates in excess of 1000 are routinely required. Such systems are sometimes referred to as channelizers [128]. Another application area is in $\Sigma\Delta$ data converters [129].

CIC filters are based on the fact that perfect pole/zero canceling can be achieved. This is only possible with exact integer arithmetic. Both two's complement and the residue number system have the ability to support error-free arithmetic. In the case of two's complement, arithmetic is performed modulo 2^b , and, in the case of the RNS, modulo M .

An introductory case study will be used to demonstrate.

5.3.1 Single-Stage CIC Case Study

Figure 5.12 shows a first order CIC filter without decimation in 4-bit arithmetic. The filter consists of a (recursive) integrator (I-section), followed by a 4-bit differentiator or comb (C-section). The filter is realized with 4-bit values, which are implemented in two's complement arithmetic, and the values are bounded by $-8_{10} = 1000_2$ and $7_{10} = 0111_2$.

Figure 5.13 shows the impulse response of the filter. Although the filter is recursive, the impulse response is finite, i.e., it is a *recursive FIR* filter. This

is unusual because we generally expect a recursive filter to be an IIR filter. The impulse response shows that the filter computes the sum

$$y[n] = \sum_{k=0}^{D-1} x[n-k], \quad (5.20)$$

where D is the delay found in the comb section. The filter's response is a *moving average* defined over D contiguous sample values. Such a moving average is a very simple form of a lowpass filter. The same moving-average filter implemented as a nonrecursive FIR filter, would require $D - 1 = 5$ adders, compared with one adder and one subtractor for the CIC design.

A recursive filter having a known pole location has its largest steady-state sinusoidal output when the input is an “eigenfrequency” signal, one whose pole directly coincides with a pole of the recursive filter. For the CIC section, the eigenfrequency corresponds to the frequency $\omega = 0$, i.e., a step input. The step response of the first order moving average given by (5.20) is a ramp for the first D samples, and a constant $y[n] = D = 6$ thereafter, as shown in Fig. 5.14. Note that although the integrator $w[n]$ shows frequent overflows, the output is still correct. This is because the comb subtraction also uses two's complement arithmetic, e.g., at the time of the first wrap-around, the actual integrator signal is $w[n] = -8_{10} = 1000_2C$, and the delay signal is $w[n-6] =$

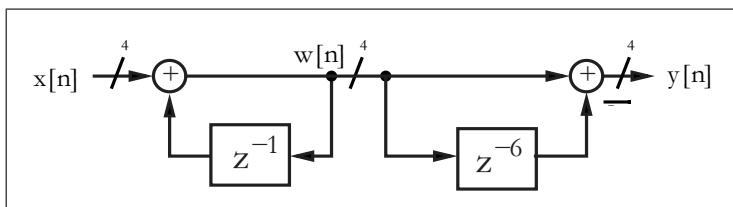


Fig. 5.12. Moving average in 4-bit arithmetic

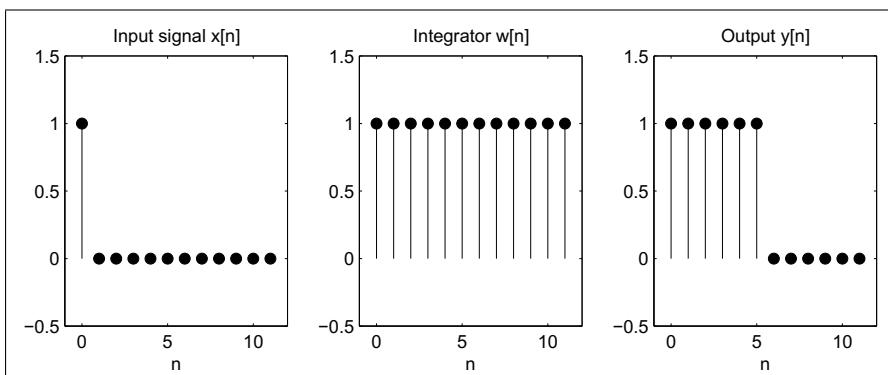
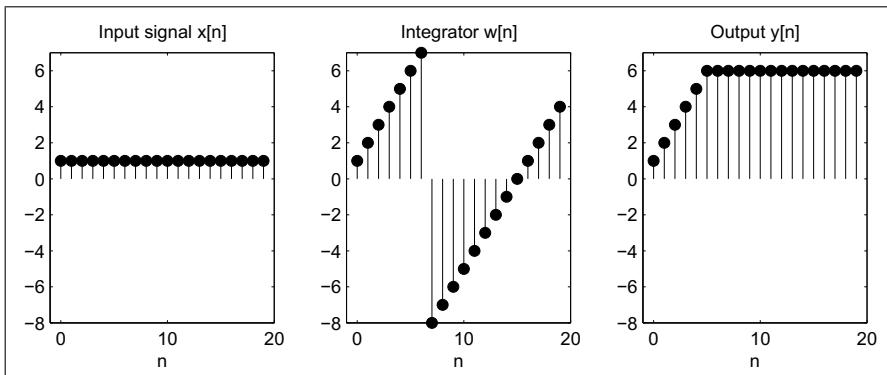


Fig. 5.13. Impulse response of the filter from Fig. 5.12

Table 5.2. RNS mapping for the set $\{2, 3, 5\}$

$a =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$a \bmod 2$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$a \bmod 3$	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0
$a \bmod 5$	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0
$a =$	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	
$a \bmod 2$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0
$a \bmod 3$	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	
$a \bmod 5$	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	

$2_{10} = 0010_2C$. This results in $y[n] = -8_{10} - 2_{10} = 1000_2C - 0010_2C = 0110_2C = 6_{10}$, as expected. The accumulator would continue to count upward until $w[n] = -8_{10} = 1000_2C$ is again reached. This pattern would continue as long as the step input is present. In fact, as long as the output $y[n]$ is a valid 4-bit two's complement number in the range $[-8, 7]$, the exact arithmetic of the two's complement system will automatically compensate for the integrator overflows.

**Fig. 5.14.** Step response (eigenfrequency test) of the filter from Fig. 5.12

In general, a 4-bit filter width is usually much too small for a typical application. The Harris IC HSP43220, for instance, has five stages and uses a 66-bit integrator width. To reduce the adder latency, it is therefore reasonable to use a multibase RNS system. If we use, for instance, the set $\mathbb{Z}_{30} = \{2, 3, 5\}$, it can be seen from Table 5.2 that a total of $2 \times 3 \times 5 = 30$ unique values can be represented. The mapping is unique (bijective) and is proven by the Chinese remainder theorem.

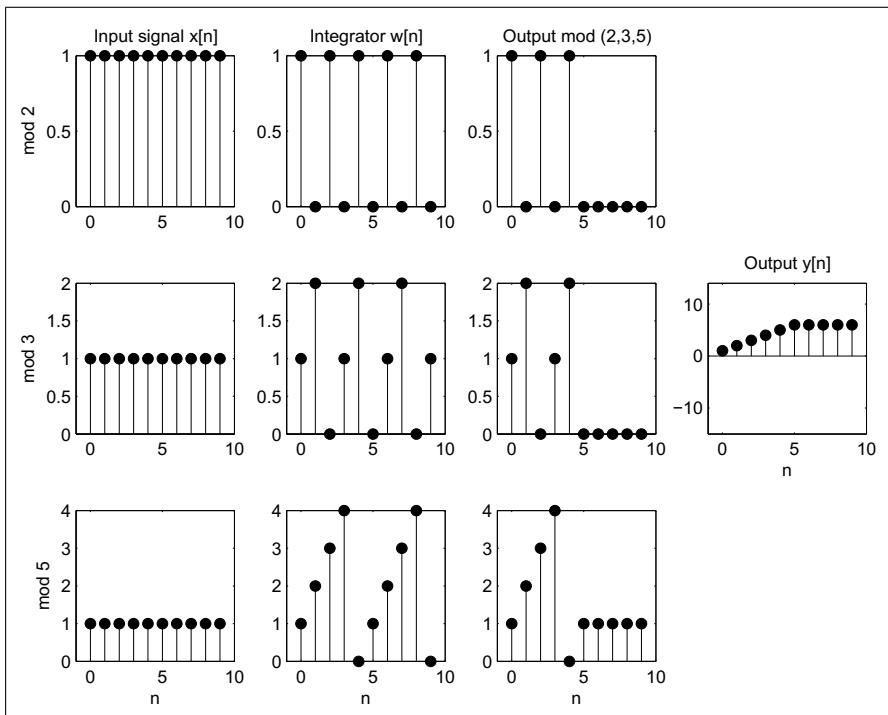


Fig. 5.15. Step response of the first order CIC in RNS arithmetic

Figure 5.15 displays the step response of the illustrated RNS implementation. The filter's output, $y[n]$, has been reconstructed using data from Table 5.2. The output response is identical with the sample value obtained in the two's complement case (see Fig. 5.14). A mapping that preserves the structure is called a *homomorphism*. A bijective homomorphism is called an *isomorphism* (notation \cong), which can be expressed as:

$$\mathbb{Z}_{30} \cong \mathbb{Z}_2 \times \mathbb{Z}_3 \times \mathbb{Z}_5. \quad (5.21)$$

5.3.2 Multistage CIC Filter Theory

The transfer function of a general CIC system consisting of S stages is given by:

$$F(z) = \left(\frac{1 - z^{-RD}}{1 - z^{-1}} \right)^S, \quad (5.22)$$

where D is the number of delays in the comb section, and R the downsampling (decimation) factor.

It can be seen from (5.22) that $F(z)$ is defined with respect to RDS zeros and S poles. The RD zeros generated by the numerator term $(1 - z^{-RD})$

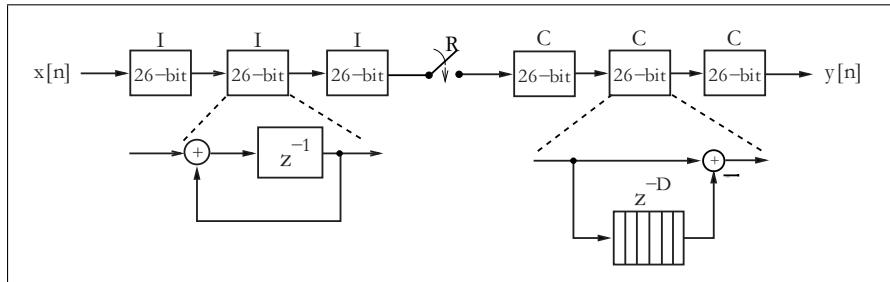


Fig. 5.16. CIC filter. Each stage 26 bits

are located on $2\pi/(RD)$ -radian centers beginning at $z = 1$. Each distinct zero appears with multiplicity S . The S poles of $F(z)$ are located at $z = 1$, i.e., at the zero frequency (DC) location. It can immediately be seen that they are annihilated by S zeros of the CIC filter. The result is an S -stage moving average filter. The maximum dynamic range growth occurs at the DC frequency (i.e., $z = 1$). The maximum dynamic range growth is

$$B_{\text{grow}} = (RD)^S \quad \text{or} \quad b_{\text{grow}} = \log_2(B_{\text{grow}}) \text{ bits.} \quad (5.23)$$

Knowledge of this value is important when designing a CIC filter, since the need for exact arithmetic as shown in the single-state CIC example. In practice, the worst-case gain can be substantial, as evidenced by a 66-bit dynamic range built into commercial CIC filters (e.g., the Harris HSP43220 [128] channelizer), typically designed using two's complement arithmetic.

Figure 5.16 shows a three-stage CIC filter that consists of a three-stage integrator, a sampling rate reduction by R , and a three-stage comb. Note that all integrators are implemented first, then the decimator, and finally the comb sections. The rearrangement saves a factor R of delay elements in the comb sections. The number of delays D for a high-decimation rate filter is typically one or two.

A three-stage CIC filter with an input wordwidth of eight bits, along with $D = 2$, $R = 32$, or $DR = 2 \times 32 = 64$, would require an internal wordwidth of $W = 8 + 3 \log_2(64) = 26$ bits to ensure that run-time overflow would not occur. The output wordwidth would normally be a value significantly less than W , such as 10 bits.

Example 5.4: Three-Stage CIC Decimator I

The worst-case gain condition can be forced by supplying a step (DC) signal to the CIC filter. Fig. 5.17a shows a step input signal with amplitude 127. Figure 5.17b displays the output found at the third integrator section. Observe that run-time overflows occur at a regular rate. The CIC output shown in Fig. 5.17c is interpolated (smoothed) for display at the input sampling rate. The output shown in Fig. 5.17d is scaled to 10-bit precision and displayed at the decimated sample rate.

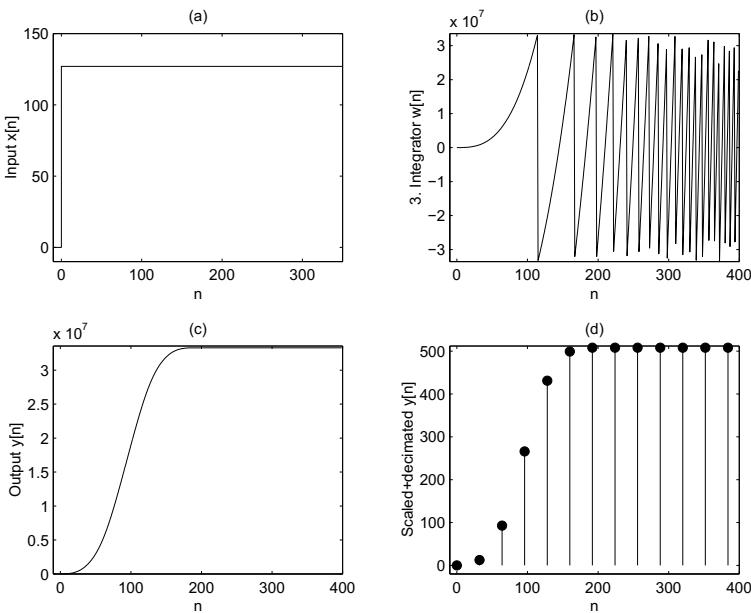


Fig. 5.17. MATLAB simulation of the three-stage CIC filter shown in Fig. 5.16

The following VHDL code⁵ shows the CIC example design:

```

LIBRARY ieee; USE ieee.std_logic_1164.ALL;

PACKAGE n_bit_int IS
  SUBTYPE U5 IS INTEGER RANGE 0 TO 32;
  SUBTYPE SLV8 IS STD_LOGIC_VECTOR(7 DOWNTO 0);
  SUBTYPE SLV10 IS STD_LOGIC_VECTOR(9 DOWNTO 0);
  SUBTYPE SLV26 IS STD_LOGIC_VECTOR(25 DOWNTO 0);
END n_bit_int;

LIBRARY work;
USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;

-----
```

```

ENTITY cic3r32 IS
  PORT (clk      : IN STD_LOGIC; -- System clock
        reset   : IN STD_LOGIC; -- Asynchronous reset
        x_in    : IN SLV8;      -- System input
        clk2    : OUT STD_LOGIC; -- Clock divider
        y_out   : OUT SLV10);  -- System output
```

⁵ The equivalent Verilog code `cic3r32.v` for this example can be found in Appendix A on page 831. Synthesis results are shown in Appendix B on page 881.

```

END cic3r32;
-----
ARCHITECTURE fpga OF cic3r32 IS

SUBTYPE SLV26 IS STD_LOGIC_VECTOR(25 DOWNTO 0);

TYPE STATE_TYPE IS (hold, sample);
SIGNAL state : STATE_TYPE ;
SIGNAL count : U5;
SIGNAL x : SLV8; -- Registered input
SIGNAL sxtx : SLV26; -- Sign extended input
SIGNAL i0, i1 , i2 : SLV26; -- I section 0, 1, and 2
SIGNAL i2d1, i2d2, c1, c0 : SLV26; -- I and COMB section 0
SIGNAL c1d1, c1d2, c2 : SLV26;-- COMB1
SIGNAL c2d1, c2d2, c3 : SLV26;-- COMB2

BEGIN

FSM: PROCESS (reset, clk)
BEGIN
    IF reset = '1' THEN -- Asynchronous reset
        state <= hold;
        count <= 0;
        clk2 <= '0';
    ELSIF rising_edge(clk) THEN
        IF count = 31 THEN
            count <= 0;
            state <= sample;
            clk2 <= '1';
        ELSE
            count <= count + 1;
            state <= hold;
            clk2 <= '0';
        END IF;
    END IF;
END PROCESS FSM;

sxt: PROCESS (x)
BEGIN
    sxtx(7 DOWNTO 0) <= x;
    FOR k IN 25 DOWNTO 8 LOOP
        sxtx(k) <= x(x'high);
    END LOOP;
END PROCESS sxt;

Int: PROCESS(clk, reset)
BEGIN
    IF reset = '1' THEN -- Asynchronous clear
        x <= (OTHERS => '0'); i0 <= (OTHERS => '0');
        i1 <= (OTHERS => '0'); i2 <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
        x      <= x_in;
    END IF;
END PROCESS Int;

```

```

        i0    <= i0 + sxtx;
        i1    <= i1 + i0 ;
        i2    <= i2 + i1 ;
    END IF;
END PROCESS Int;

Comb: PROCESS(clk, reset, state)
BEGIN
    IF reset = '1' THEN -- Asynchronous clear
        c0 <= (OTHERS => '0'); c1 <= (OTHERS => '0');
        c2 <= (OTHERS => '0'); c3 <= (OTHERS => '0');
        i2d1 <= (OTHERS => '0'); i2d2 <= (OTHERS => '0');
        c1d1 <= (OTHERS => '0'); c1d2 <= (OTHERS => '0');
        c2d1 <= (OTHERS => '0'); c2d2 <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
        IF state = sample THEN
            c0    <= i2;
            i2d1 <= c0;
            i2d2 <= i2d1;
            c1    <= c0 - i2d2;
            c1d1 <= c1;
            c1d2 <= c1d1;
            c2    <= c1 - c1d2;
            c2d1 <= c2;
            c2d2 <= c2d1;
            c3    <= c2 - c2d2;
        END IF;
    END IF;
END PROCESS Comb;

y_out <= c3(25 DOWNTO 16); -- i.e., c3 / 2**16

```

END fpga;

The designed filter includes a finite state machine (FSM), a sign extension, `sxt`: PROCESS, and two arithmetic PROCESS blocks. The `FSM`: PROCESS contains the clock divider for the comb section. The `Int`: PROCESS realizes the three integrators. The `Comb`: PROCESS includes the three comb filters, each having a delay of two samples. The filter uses 341 LEs, no embedded multiplier, and has an `Fmax`=282.49 MHz registered performance using the TimeQuest slow 85C model. Note that the filter would require many more LEs without the early downsampling. The early downsampling saves $3 \times 32 \times 26 = 2496$ registers or LEs.

If we compare the filter outputs (Fig. 5.18 shows the VHDL output `y_out`, and the response $y[n]$ from the MATLAB simulation shown in Fig. 5.17d we see that the filter behaves as expected.

5.4

Hogenauer [127] noted, based on a careful analysis, that some of the lower significant bits from early stages can be eliminated without sacrificing system integrity. Figure 5.19 displays the system's magnitude frequency response for a design using full (worst-case) wordwidth in all stages, and using the wordlength "pruning" policy suggested by Hogenauer.

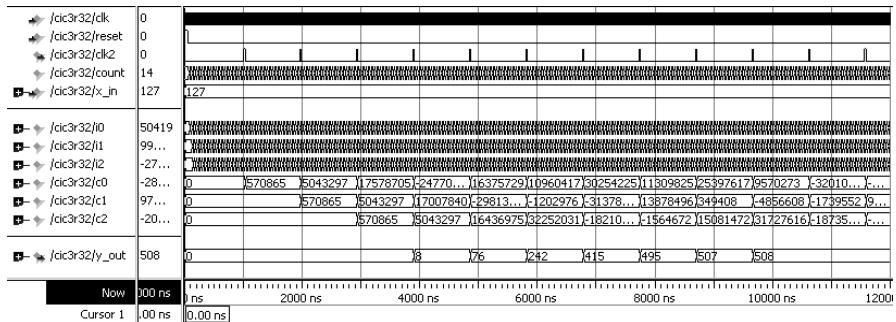


Fig. 5.18. VHDL simulation of the three-stage CIC filter shown in Fig. 5.16

5.3.3 Amplitude and Aliasing Distortion

The transfer function of an S -stage CIC filter was reported to be

$$F(z) = \left(\frac{1 - z^{-RD}}{1 - z^{-1}} \right)^S. \quad (5.24)$$

The amplitude distortion and the maximum aliasing component can be computed in the frequency domain by evaluating $F(z)$ along the arc $z = e^{j2\pi fT}$. The magnitude response becomes

$$|F(f)| = \left(\frac{\sin(2\pi fTRD/2)}{\sin(2\pi fT/2)} \right)^S, \quad (5.25)$$

which can be used to directly compute the *amplitude distortion* at the passband edge ω_p . Figure 5.20 shows $|F(f - k\frac{1}{2R})|$ for a three-stage CIC filter with $R = 3$, $D = 2$, and $RD = 6$. Observe that several copies of the CIC filter's low-frequency response are aliased in the baseband.

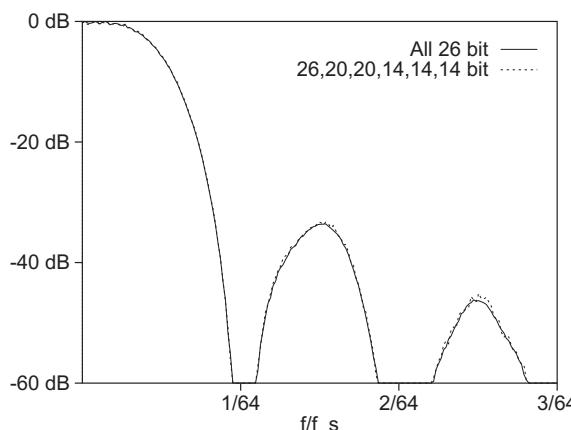


Fig. 5.19. CIC transfer function (f_s is sampling frequency at the input)

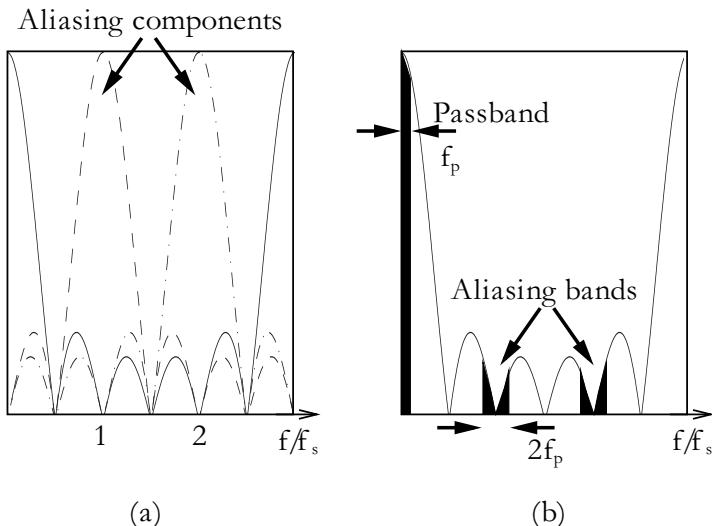


Fig. 5.20. Transfer function of a three-stage CIC decimator. Note that f_s is the sampling frequency at the lower rate

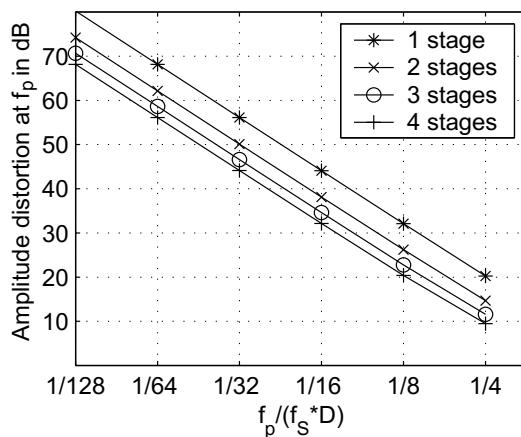


Fig. 5.21. Amplitude distortion $-20 \log_{10}(1 - F(f_p))$ for the CIC decimator

It can be seen that the maximum aliasing component can be computed from $|F(f)|$ at the frequency

$$f|_{\text{Aliasing has maximum}} = 1/(2R) - f_p. \quad (5.26)$$

Most often, only the first aliasing component is taken into consideration, because the second component is smaller. Figure 5.21 shows the amplitude distortion $-20 \log_{10}(1 - F(f_p))$ at f_p for different ratios of $f_p / (Df_s)$.

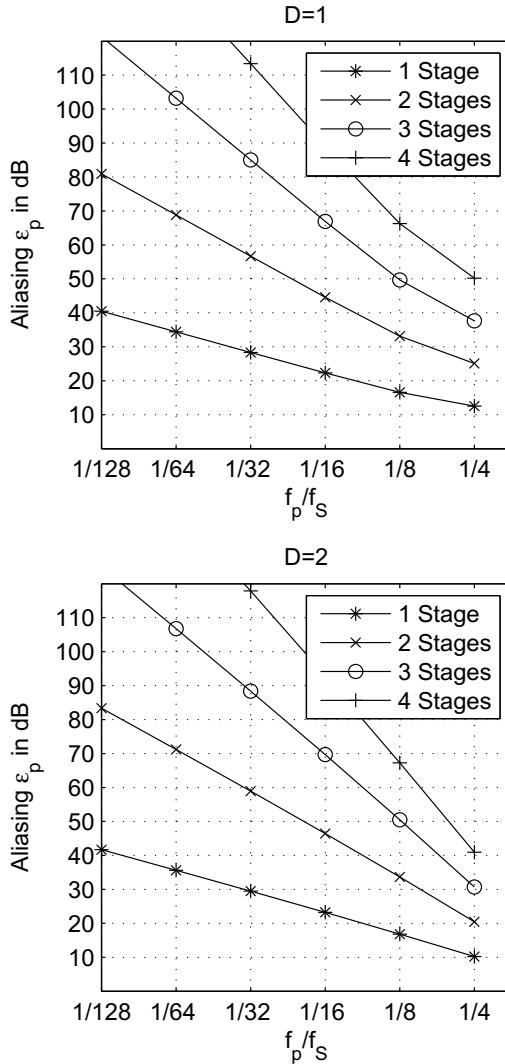


Fig. 5.22. Maximum aliasing for one- to four stage CIC decimator

Figure 5.22 shows, for different values of S , R , and D , the maximum aliasing component for a special ratio of passband frequency and sampling frequency, f_p/f_s .

It may be argued that the amplitude distortion can be corrected with a cascaded FIR compensation filter, which has a transfer function $1/|F(z)|$ in the passband, but the aliasing distortion can *not* be repaired. Therefore, the acceptable aliasing distortion is most often the dominant design parameter.

5.3.4 Hogenauer Pruning Theory

The total internal wordwidth is defined as the sum of the input wordwidth and the maximum dynamic growth requirement (5.23), or algebraically:

$$B_{\text{intern}} = B_{\text{input}} + B_{\text{growth}}. \quad (5.27)$$

If the CIC filter is designed to perform exact arithmetic with this wordwidth at all levels, no run-time overflow will occur at the output. In general, input and output bit width of a CIC filter are in the same range. We find then that quantization introduced through pruning in the output is, in general, larger than quantization introduced by also pruning some LSBs at previous stages. If $\sigma_{T,2S+1}^2$ is the quantization noise introduced through pruning in the output, Hogenauer suggested to set it equal to the sum of the noise σ_k^2 introduced by all previous sections. For a CIC filter with S integrator and S comb sections, it follows that:

$$\sum_{k=1}^{2S} \sigma_{T,k}^2 = \sum_{k=1}^{2S} \sigma_k^2 P_k^2 \leq \sigma_{T,2S+1}^2 \quad (5.28)$$

$$\sigma_{T,k}^2 = \frac{1}{2S} \sigma_{T,2S+1}^2 \quad (5.29)$$

$$P_k^2 = \sum_n (h_k[n])^2 \quad k = 1, 2, \dots, 2S, \quad (5.30)$$

where P_k^2 is the power gain from stage k to the output. Compute next the number of bits B_k , which should be pruned by

$$B_k = \left\lceil 0.5 \log_2 \left(P_k^{-2} \times \frac{6}{N} \times \sigma_{T,2S+1}^2 \right) \right\rceil \quad (5.31)$$

$$\sigma_{T,k}^2|_{k=2S+1} = \frac{1}{12} 2^{2B_k} = \frac{1}{12} 2^{2(B_{\text{in}} - B_{\text{out}} + B_{\text{growth}})}. \quad (5.32)$$

The power gain P_k^2 , $k = S+1, \dots, 2S$ for the comb sections can be computed using the binomial coefficient

$$H_k(z) = \sum_{n=0}^{2S+1-k} (-1)^n \binom{2S+1-k}{n} z^{-kRD} \\ k = S, S+1, \dots, 2S. \quad (5.33)$$

For computation of the first factor P_k^2 for $k = 1, 2, \dots, S$, it is useful to keep in mind that each integrator/comb pair produces a finite (moving average) impulse response. The resulting system for stage k is therefore a series of $S-k+1$ integrator/comb pairs followed by $k-1$ comb sections. Figure 5.23 shows this rearrangement for a simplified computation of P_k^2 .

The program `cic.exe` (included on the CD-ROM under `util`) computes this CIC pruning. The program produces the impulse response `cicXX.imp` and a configuration file `cicXX.dat`, where `XX` must be specified. The following design example explains the results.

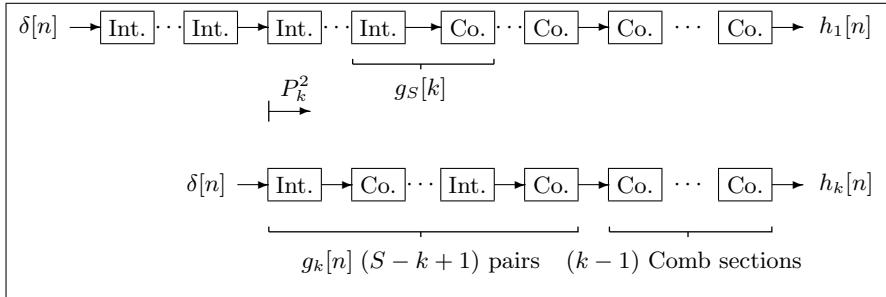


Fig. 5.23. Rearrangement to simplify the computation of P_k^2 . (© VDI Press [4])

Example 5.5: Three-Stages CIC Decimator II

Let us design the same overall CIC filter as in Example 5.4 (p. 321) but this time with bit pruning. The raw data of the decimator were: $B_{\text{input}} = 8$, $B_{\text{output}} = 10$, Bit $R = 32$, and $D = 2$. Obviously, the bit growth is

$$B_{\text{growth}} = \lceil \log_2(RD^S) \rceil = \log_2(64^3) \lceil 3 \times 6 \rceil = 18, \quad (5.34)$$

and the total internal bit width becomes

$$B_{\text{intern}} = B_{\text{input}} + B_{\text{growth}} = 8 + 18 = 26. \quad (5.35)$$

The program `cic.exe` shows the following results:

```
-- Program for the design of a CIC decimator.
-----
-- Input bit width      Bin   =    8
-- Output bit width     Bout  =   10
-- Number of stages     S     =    3
-- Decimation factor    R     =   32
-- COMB delay           D     =    2
-- Frequency resolution DR    =   64
-- Passband freq. ratio P     =    8
-----
----- Results of the Design -----
-----
----- Computed bit width:
----- Maximum bit growth over all stages      =    18
----- Maximum bit width including sign Bmax+1 =    26
-- Stage 1 INTEGRATOR. Bit width :  26
-- Stage 2 INTEGRATOR. Bit width :  21
-- Stage 3 INTEGRATOR. Bit width :  16
-- Stage 1 COMB.        Bit width :  14
-- Stage 2 COMB.        Bit width :  13
-- Stage 3 COMB.        Bit width :  12
----- Maximum aliasing component : 0.002135 = 53.41 dB
----- Amplitude distortion       : 0.729769 =  2.74 dB
```

5.5

The design charts shown in Figs. 5.21 and 5.22 may also be used to compute the maximum aliasing component and the amplitude distortion. If we

compare this data with the tables provided by Hogenauer then the aliasing suppression is 53.4 dB (for Delay = 2 [127, Table II]), and the passband attenuation is 2.74 dB [127, Table I]. Hogenauer and `cic.exe` compute amplitude distortion in dB via $-20 \log_{10}(F(f_p))$ while the data in Table 5.21 follow typical textbooks, i.e., compute $-20 \log_{10}(1 - F(f_p))$. Note that the Table I provided by Hogenauer are normalized with the comb delay, while the program `cic.exe` does not normalize with the comb delay.

The following design example demonstrates the detailed bit-width design, using Quartus II:

Example 5.6: Three-Stage CIC Decimator III

The data for the design should be the same as for Example 5.4 (p. 321), but we now consider the pruning as computed in Example 5.5 (p. 329).

The following VHDL code⁶ shows the CIC example design with pruning:

```

LIBRARY ieee; USE ieee.std_logic_1164.ALL;

PACKAGE n_bit_int IS                         -- User defined type
    SUBTYPE U5 IS INTEGER RANGE 0 TO 32;
    SUBTYPE SLV8  IS STD_LOGIC_VECTOR(7 DOWNTO 0);
    SUBTYPE SLV10 IS STD_LOGIC_VECTOR(9 DOWNTO 0);
    SUBTYPE SLV12 IS STD_LOGIC_VECTOR(11 DOWNTO 0);
    SUBTYPE SLV13 IS STD_LOGIC_VECTOR(12 DOWNTO 0);
    SUBTYPE SLV14 IS STD_LOGIC_VECTOR(13 DOWNTO 0);
    SUBTYPE SLV16 IS STD_LOGIC_VECTOR(15 DOWNTO 0);
    SUBTYPE SLV21 IS STD_LOGIC_VECTOR(20 DOWNTO 0);
    SUBTYPE SLV26 IS STD_LOGIC_VECTOR(25 DOWNTO 0);
END n_bit_int;

LIBRARY work; USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-----
ENTITY cic3s32 IS
    PORT (clk      : IN STD_LOGIC; -- System clock
          reset   : IN STD_LOGIC; -- Asynchronous reset
          x_in    : IN SLV8;      -- System input
          clk2    : OUT STD_LOGIC; -- Clock divider
          y_out   : OUT SLV10);  -- System output
END cic3s32;
-----
ARCHITECTURE fpga OF cic3s32 IS

    TYPE STATE_TYPE IS (hold, sample);
    SIGNAL state   : STATE_TYPE;
    SIGNAL count   : U5;
    SIGNAL x       : SLV8;      -- Registered input

```

⁶ The equivalent Verilog code `cic3s32.v` for this example can be found in Appendix A on page 833. Synthesis results are shown in Appendix B on page 881.

```

SIGNAL  sxtx   : SLV26;      -- Sign extended input
SIGNAL  i0      : SLV26;      -- I section 0
SIGNAL  i1      : SLV21;      -- I section 1
SIGNAL  i2      : SLV16;      -- I section 2
SIGNAL  i2d1, i2d2, c1, c0 : SLV14;
                                -- I and COMB section 0
SIGNAL  c1d1, c1d2, c2 : SLV13;  --COMB 1
SIGNAL  c2d1, c2d2, c3 : SLV12;  --COMB 2

BEGIN

  FSM: PROCESS (reset, clk)
  BEGIN
    IF reset = '1' THEN          -- Asynchronous reset
      state <= hold;
      count <= 0;
      clk2 <= '0';
    ELSIF rising_edge(clk) THEN
      IF count = 31 THEN
        count <= 0;
        state <= sample;
        clk2 <= '1';
      ELSE
        count <= count + 1;
        state <= hold;
        clk2 <= '0';
      END IF;
    END IF;
  END PROCESS FSM;

  Sxt : PROCESS (x)
  BEGIN
    sxtx(7 DOWNTO 0) <= x;
    FOR k IN 25 DOWNTO 8 LOOP
      sxtx(k) <= x(x'high);
    END LOOP;
  END PROCESS Sxt;

  Int: PROCESS(clk, reset)
  BEGIN
    IF reset = '1' THEN -- Asynchronous clear
      x <= (OTHERS => '0');  i0 <= (OTHERS => '0');
      i1 <= (OTHERS => '0');  i2 <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
      x  <= x_in;
      i0 <= i0 + sxtx;
      i1 <= i1 + i0(25 DOWNTO 5);  -- i.e., i0/32
      i2 <= i2 + i1(20 DOWNTO 5);  -- i.e., i1/32
    END IF;
  END PROCESS Int;

  Comb: PROCESS(clk, reset, state)
  BEGIN

```

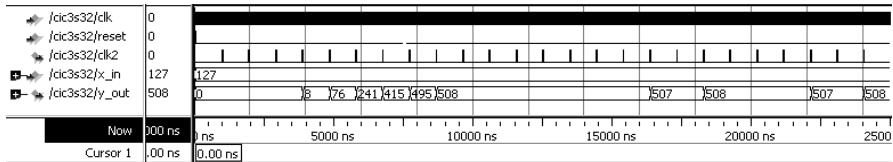


Fig. 5.24. VHDL simulation of the three-stage CIC filter, implemented with bit pruning

```

IF reset = '1' THEN -- Asynchronous clear
    c0 <= (OTHERS => '0'); c1 <= (OTHERS => '0');
    c2 <= (OTHERS => '0'); c3 <= (OTHERS => '0');
    i2d1 <= (OTHERS => '0'); i2d2 <= (OTHERS => '0');
    c1d1 <= (OTHERS => '0'); c1d2 <= (OTHERS => '0');
    c2d1 <= (OTHERS => '0'); c2d2 <= (OTHERS => '0');
ELSIF rising_edge(clk) THEN
    IF state = sample THEN
        c0 <= i2(15 DOWNTO 2);      -- i.e., i2/4
        i2d1 <= c0;
        i2d2 <= i2d1;
        c1 <= c0 - i2d2;
        c1d1 <= c1(13 DOWNTO 1);   -- i.e., c1/2
        c1d2 <= c1d1;
        c2 <= c1(13 DOWNTO 1) - c1d2;
        c2d1 <= c2(12 DOWNTO 1);   -- i.e., c2/2
        c2d2 <= c2d1;
        c3 <= c2(12 DOWNTO 1) - c2d2;
    END IF;
END IF;
END PROCESS Comb;

y_out <= c3(11 DOWNTO 2);      -- i.e., c3/4

END fpga;

```

The design has the same architecture as the unscaled CIC shown in Example 5.4 (p. 321). The design consists of a finite state machine (FSM), a sign extension `Sxt`: `PROCESS`, and two arithmetic `PROCESS` blocks. At the begin of each block a asynchronous reset is applied to all registers used in this `PROCESS`. The `FSM`: `PROCESS` contains the clock divider for the comb sections. The `Int`: `PROCESS` realizes the three integrators. The `Comb`: `PROCESS` includes the three comb sections, each having a delay of two. But now, all integrator and comb sections are designed with the bit width suggested by Hogenauer's pruning technique. This reduces the size of the design to 209 LEs and the design now runs at 290.02 MHz.

5.6

This design does not improve the speed (282.49 versus 290.02 MHz), but saves a substantial number of LEs (about 30%), compared with the design considered in Example 5.4 (p. 321). Comparing the filter output of the VHDL simulations, shown in Figs. 5.24 and 5.18 (p. 325), different LSB quantization

behavior can be noted (see Exercise 5.11, p. 412). In the pruned design, “noise” possesses the asymptotic behavior of the LSB ($507 \leftrightarrow 508$).

The design of a CIC *interpolator* and its pruning technique is discussed in Exercise 5.24, p. 415.

5.3.5 CIC RNS Design

The design of a CIC filter using the RNS was proposed by Garcia et al. [55]. A three-stage CIC filter, with 8-bit input, 10-bit output, $D = 2$, and $R = 32$ was implemented. The maximum wordwidth was 26 bits. For the RNS implementation, the 4-moduli set $(256, 63, 61, 59)$, i.e., one 8-bit two’s complement and three 6-bit moduli, covers this range (see Fig. 5.25). The output was scaled using an ε -CRT requiring eight tables and three two’s complement adders [48, Fig. 1], or (as shown in Fig. 5.26) using a base removal scaling (BRS) algorithm based on two 6-bit moduli (after [47]), and an ε -CRT for the remaining two moduli, for a total of five modulo adders and nine ROM tables, or seven tables (if multiplicative inverse ROM and the ε -CRT are combined). The following table shows the speed in MSPS and the number of LEs and EABs used for the three scaling schemes for a FLEX10K device.

Type	ε -CRT	BRS ε -CRT (Speed data for BRS m_4 only)	BRS ε -CRT combined ROM
MSPS	58.8	70.4	58.8
#LE	34	87	87
#Table (EAB)	8	9	7

The decrease in speed to 58.8 MSPS, for the scaling schemes 1 and 3, is the result of the need for a 10-bit ε -CRT. It should be noted that this does not reduce the *system* speed, since scaling is applied at the lower (output) sampling rate. For the BRS ε -CRT, it is assumed that only the BRS m_4 part (see Fig. 5.26) must run at the input sampling rate, while BRS m_3 and ε -CRT run at the output sampling rate.

Some resources can be saved if a scaling scheme, similar to Example 5.5 (p. 329), and illustrated in Fig. 5.25, is used. With this scheme, the BRS ε -CRT scheme must be applied to reduce the bit width in the earlier sections of the filter. The early use of ROMs decreases the possible throughput from 76.3 to 70.4 MSPS, which is the maximum speed of the BRS with m_4 . At the output, the efficient ε -CRT scheme was applied.

The following table summarizes the three implemented filter designs on a FLEX10K device, without including the scaling data:

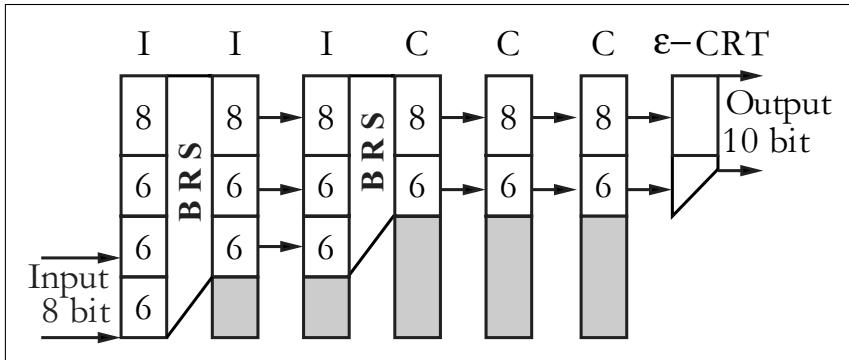


Fig. 5.25. CIC filter. Detail of design with base removal scaling (BRS)

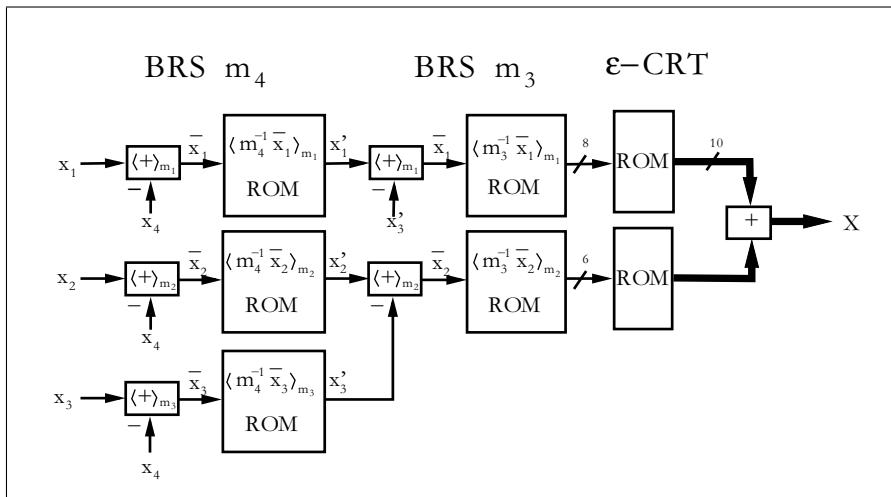


Fig. 5.26. BRS and ε -CRT conversion steps

Type	2C 26-bit	RNS 8, 6, 6, 6-bit	Detailed bit width RNS design
MSPS	49.3	76.3	70.4
#LEs	343	559	355

5.3.6 CIC Compensation Filter Design

The CIC is an efficient way to achieve high decimation or interpolation rates at little cost and high speeds. However, as we saw from the data in Fig. 5.21 (p. 326), the transfer function lacks a flat passband and a small transition band width. To improve the performance of the CIC filter the approach taken

is usually to use an FIR *compensation* filter that has a slide increasing transfer function in the passband and a much smaller transition band width. The DSP56ADC16 for instance has a four stage CIC with decimation by 16 first followed by a 255 tap FIR filter [129]. With an $f_s/f_p = 8$ the length-255 FIR allows an additional downsampling by four. Since the overall downsampling is $R_1 \times R_2 = 64$ the FIR can easily be implemented with a single MAC such as shown in Fig. 2.38, p. 126. As another example you may like to study the HSP50214 downsample, that has an I/Q splitter followed by a fifth order CIC ($R_1 = 4 \dots 32$), one to five half-band filters ($R_2 = 1 \dots 32$,) and a 255 tap CIC compensation FIR filter ($R_3 = 1 \dots 16$). Overall the decimation rate is between 4 and 16,384 [130]. As CIC interpolator we can find the data sheet of the GC4114 that has a four stage interpolator and allows up-sampling between 8 and 16,384 [131]. The GC4114 has a CFIR compensation filter with 31 constant coefficients we studied in Exercises 3.9 (p. 221) and will be designed in Exercise 5.25 (p. 416).

The design of CIC compensation filters is not too complicated since we only design an FIR filter. We compute the transfer function of the CIC filter first (see (5.25)), build the reciprocal in the frequency domain for the range of frequencies we need for the compensation, and set the other values to zero. Then we can use the inverse DFT method (see (3.12), p. 187) or if we have MATLAB the `fir2` function that works with a frequency sampling-based approach. In principle we are free in the choice of the sampling frequency to passband ratio f_s/f_p . However, we should remember if we make it too large, then a longer filter will be required running at a higher rate. If we make it too small than the large drop in the CIC amplitude will make the compensation filter design more difficult since the CIC has dropped substantially at the passband edge. A good compromise is the $f_s/f_p = 8$ ratio that allows a final reduction $R = 4$ and reasonable filter length. For our example CIC amplitude drops to 0.727, i.e., our compensation filter need to be 1.37 at the passband edge. $f_s/f_p = 8$ was not only used in the specification from Example 5.5, p. 329 but also by the DSP56ADC16 [129].

For our `cic3r32` filter specification we can use the following MATLAB script to compute the filter coefficient as well as the overall transfer function of the CIC and compensation filter cascade:

```

close all; clear all
%%%%% CIC filter parameters from Example 5.5
R = 32; D = 2; S = 3;
L = 255-1; % Filter order; fir2 requires even order
Fs = R; %% (High) Sampling freq in Hz before decimation
Fp = 1/8; %% Pass band edge
Fo = R*Fp/Fs; %% Normalized cutoff frequency
cic=[1];
for k=1:S    %% Compute the CIC impulse response
    cic=conv(cic,ones(1,R*D));

```

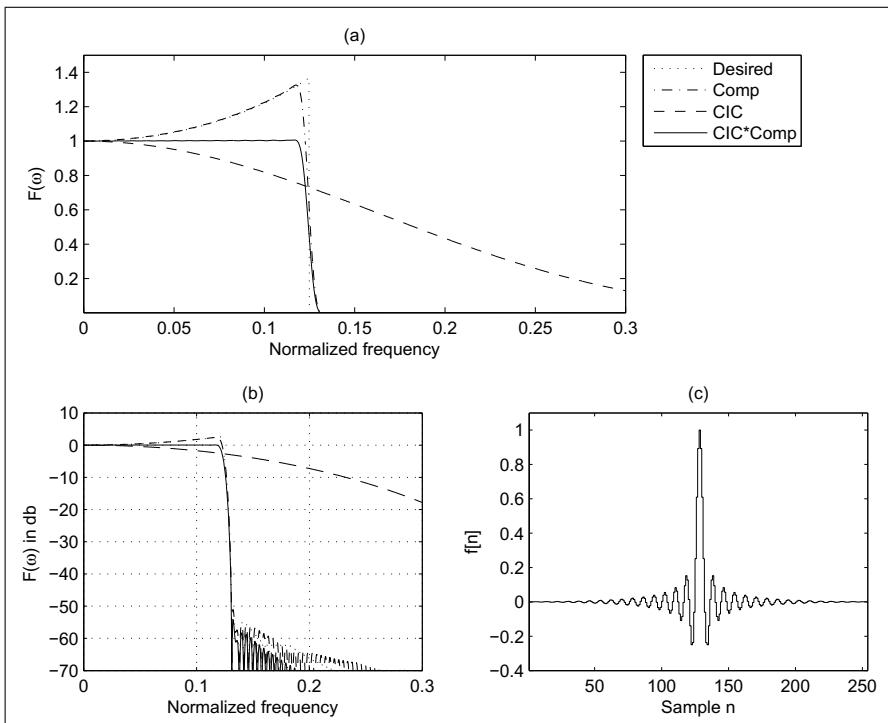


Fig. 5.27. CIC compensation filter design for filter length 255

```

end
p = 2^16; %% Sample points
[H,w] = freqz(cic,[1],p,Fs); %% Compute CIC spectrum
N=length(find(w<Fo)); %% Take all less than Fo
f = [w(1:N)' Fo .5]*2; %% Frequency sample points
H=abs(H');H=H./H(1); %% Normalize amplitudes
Mf = [1./H(1:N) 0 0]; %% Inverse CIC is desired then 0
h = fir2(L,f,Mf); %% Filter order, freq. and values
h = h/max(h); %% Normalized filter coefficients

%%%%% Compare and plot the results
figure
[H2,w2] = freqz(h,[1],p,Fs/R); %% 2. filter with 1/R rate
H2=abs(H2);H2=H2./H2(1);
H1d=H(1:p/R)'; H2d=H2(1:R:p); %% Pick same length and scale
H12=H1d.*H2d; %% Compute product filter
w12=w(1:p/R);
subplot(211)
plot(f/2*Fs/R,Mf,'k:',w2,H2,'k-.',w(1:p),H(1:p),...

```

```

'k--',w12,H12,'k-')
legend('Desired','Comp','CIC','CIC*Comp',-1);title('(a)')
ylabel('F(\omega)'); xlabel('Normalized frequency');
axis([0 .3 0.001 1.5]);
subplot(223)
plot(f/2*Fs/R,20*log10(Mf),'k:',w2,20*log10(H2),'k-.',...
      w,20*log10(H),'k--',w12,20*log10(H12),'k-')
axis([0 .3 -70 10]); grid on; title('(b)')
ylabel('F(\omega) in db'); xlabel('Normalized frequency');

subplot(224)
stairs(h); title('(c)')%% plot filter coefficients
axis([1 L+1 -.4 1.1]); ylabel('f[n]'); xlabel('Sample n')
print -deps cicomp.eps; print -djpeg90 cicomp.jpg

```

Figure 5.27 shows the results for the filter design for a length-255 filter. From the script we note first the specification of the data from Example 5.5, p. 329 followed by the computation of the CIC impulse response via S convolutions. The CIC transfer function can be computed directly via (5.25) as in the Altera application note [132] or with the `freqz` function. The routine `fir2` will then deliver the filter coefficients. Finally we plot the four transfer functions in the frequency domain. First comes the desired compensation filter functions followed by the actual result of the compensation filter. The third function is the CIC transfer function and finally the result for the cascade of the two filters is computed and shown. The same plot is done in Fig. 5.27b using logarithmic scale to see the stop band attenuation better. We can see clearly the passband frequency at $1/8=0.125$ of the lower sampling frequency. Finally the impulse response of the compensation filter is given. The only thing we need to be a little careful with is the different scaling in the frequency due to the sampling rate reduction. We can also see from Fig. 5.27 that desired and actual compensation filters are quite close since we have a substantial long filter with 255 coefficients. If we use a shorter filter then the transition band will become wider as Fig. 5.28 shows for a filter of length 31.

5.4 Multistage Decimator

If the decimation rate R is large it can be shown that a multistage design can be realized with less effort than a single-stage converter. In particular, S stages, each having a decimation capability of R_k , are designed to have an overall downsampling rate of $R = R_1 R_2 \cdots R_S$. Unfortunately, passband imperfections, such as ripple deviation, accumulate from stage to stage. As a result, a passband deviation target of ε_p must normally be tightened on the order of $\varepsilon'_p = \varepsilon_p/S$ to meet overall system specifications. This is obviously a worst-case assumption, in which all short filters have the maximum ripple at

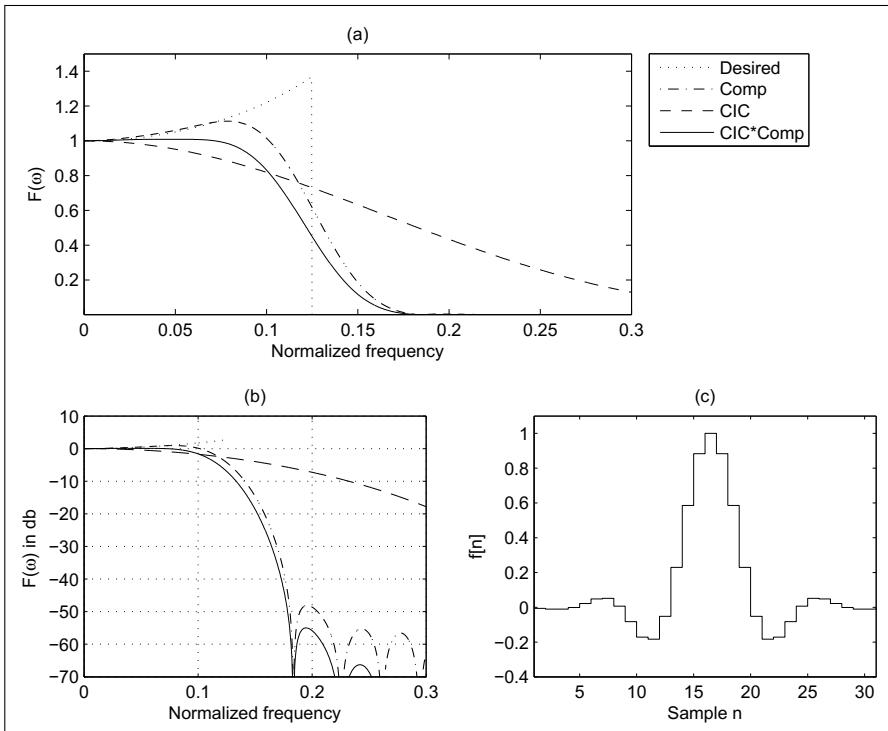


Fig. 5.28. CIC compensation filter design for filter length 31

the same frequencies, which is, in general, too pessimistic. It is often more reasonable to try an initial value near the given passband specification ε_p , and then selectively reduce it if necessary.

5.4.1 Multistage Decimator Design Using Goodman–Carey Half-Band Filters

Goodman and Carey [89] proposed to develop multistage systems based on the use of CIC and half-band filters. As the name implies, a *half-band filter* has a passband and stopband located at $\omega_s = \omega_p = \pi/2$, or midway in the baseband. A half-band filter can therefore be used to change the sampling rate by a factor of two. If the half-band filter has *point symmetry* relative to $\omega = \pi/2$, then all even coefficients (except the center tap) become zero.

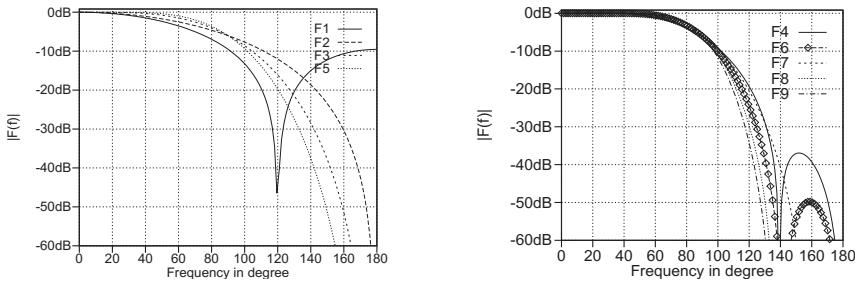


Fig. 5.29. Transfer function of the half-band filter F1 to F9

Definition 5.7: Half-Band Filter

The centered impulse response of a half-band filter obeys the following rule:

$$f[k] = 0 \quad k = \text{even without } k = 0. \quad (5.36)$$

The same condition transformed in the z -domain reads

$$F(z) + F(-z) = c, \quad (5.37)$$

where $c \in \mathbb{C}$. For a causal half-band filter this condition translates to

$$F(z) - F(-z) = cz^{-d}, \quad (5.38)$$

since now all (except one) odd coefficients are zero.

Goodman and Carey [89] have compiled a list of integer half-band filters that, with increased length, have smaller amplitude distortions. Table 5.3 shows the coefficients of these half-band filters. To simplify the representation, the coefficients were noted with a center tap located at $d = 0$. F1 is the moving-average filter of length L , i.e., it is Hogenauer's CIC filter, and may therefore be used in the first stage also, to change the rate with a factor other than two. Figure 5.29 shows the transfer function of the nine different filters. Note that in the logarithmic plot of Fig. 5.29, the point symmetry (as is usual for half-band filters) cannot be observed.

Table 5.3. Centered coefficients of the half-band filter F1 to F9 from Goodman and Carey [89]

Name	L	Ripple	$f[0]$	$f[1]$	$f[3]$	$f[5]$	$f[7]$	$f[9]$
F1	3	—	1	1				
F2	3	—	2	1				
F3	7	—	16	9	-1			
F4	7	36 dB	32	19	-3			
F5	11	—	256	150	-25	3		
F6	11	49 dB	346	208	-44	9		
F7	11	77 dB	512	302	-53	7		
F8	15	65 dB	802	490	-116	33	-6	
F9	19	78 dB	8192	5042	-1277	429	-116	18

The basic idea of the Goodman and Carey multistage decimator design is that, in the first stages, filters with larger ripple and less complexity can be applied, because the passband-to-sampling frequency ratio is relatively small. As the passband-to-sampling frequency ratio increases, we must use filters with less distortion. The algorithm stops at $R = 2$. For the final decimation ($R = 2$ to $R = 1$), a longer half-band filter must be designed.

Goodman and Carey have provided the design chart shown in Fig. 5.30. Initially, the input oversampling factor R and the necessary attenuation in the passband and stopband $A = A_p = A_s$ must be computed. From this starting point, the necessary filters for $R, R/2, R/4, \dots$ can be drawn as a horizontal line (at the same stopband attenuation). The filters F4 and F6–F9 have ripple in the passband (see Exercise 5.8, p. 412), and if several such filters are used it may be necessary to adjust ε_p . We may, therefore, consider the following adjustment:

$$A = -20 \log_{10} \varepsilon_p \quad \text{for } F1\text{--}F3, F5 \quad (5.39)$$

$$A = -20 \log_{10} \min \left(\frac{\varepsilon_p}{S'}, \varepsilon_s \right) \quad \text{for } F4, F6\text{--}F9, \quad (5.40)$$

where S' is the number of stages with ripple.

We will demonstrate the multistage design with the following example.

Example 5.8: Multistage Half-Band Filter Decimator

We wish to develop a decimator with $R = 160$, $\varepsilon_p = 0.015$, and $\varepsilon_s = 0.031 = 30$ dB, using the Goodman and Carey design approach.

At first glance, we can conclude that we need a total of five filters and mark the starting point at $R = 160$ and 30 dB in Fig. 5.31a. From 160 to 32, we use a CIC filter of length $L = 5$. This CIC filter is followed by two F2 filter and one F3 filter to reach $R = 8$. Now we need a filter with ripple. It follows that:

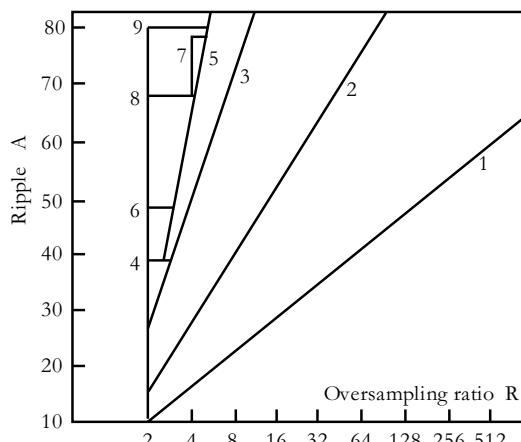


Fig. 5.30. Goodman and Carey design chart [89]

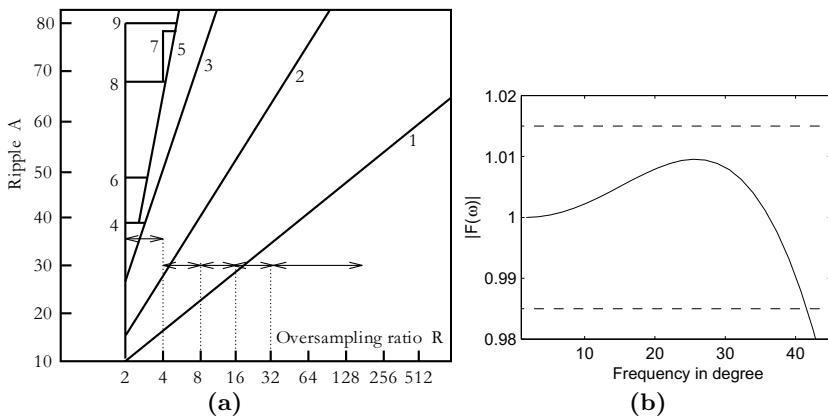


Fig. 5.31. Design example for Goodman and Carey half-band filter. (a) Design chart. (b) Transfer function $|F(\omega)|$

$$A = -20 \log_{10} \min \left(\frac{0.015}{1}, 0.031 \right) = 36.48 \text{ dB.} \quad (5.41)$$

From Fig. 5.30, we conclude that for 36 dB the filter F4 is appropriate. Figure 5.31a shows the design algorithm, using the design chart from Fig. 5.30. We may now compute the whole filter transfer function $|F(\omega)|$ by using the Noble relation (see Fig. 5.6, p. 309) $F(z) = F_1(z)F_2(z^5)F_2(z^{10})F_3(z^{20})F_4(z^{40})$, who's passband is shown in Fig. 5.31b.

5.8

Example 5.8 shows that considering only the filter with ripple in (5.40) was sufficient. Using a more pessimistic approach, with $S = 6$, we would have obtained $A = -20 \log(0.015/6) = 52$ dB, and we would have needed filter F8, with essentially higher effort. It is therefore better to start with an optimistic assumption and possibly correct this later.

5.5 Frequency-Sampling Filters as Bandpass Decimators

The CIC filters discussed in Sect. 5.3 (p. 317) belong to a larger class of systems called *frequency-sampling filters* (FSFs). Frequency-sampling filters can be used, as channelizer or decimating filter, to decompose the information spectrum into a set of discrete subbands, such as those found in multiuser communication systems. A classic FSF consists of a comb filter cascaded with a bank of frequency-selective resonators [4, 73]. The resonators independently produce a collection of poles that selectively annihilate the zeros produced by the comb prefilter. Gain adjustments are applied to the output of the resonators to shape the resulting magnitude frequency response of the overall

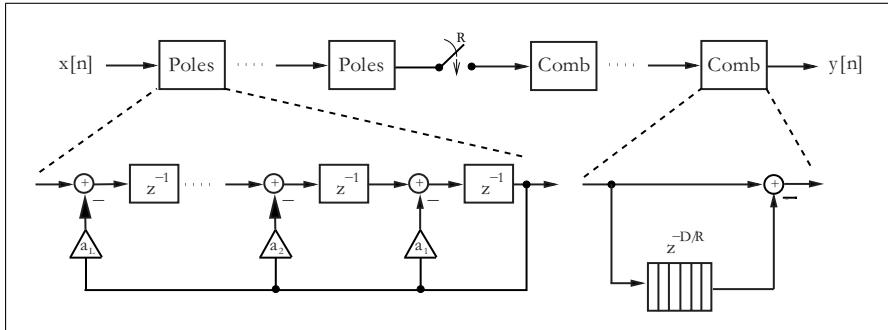


Fig. 5.32. Cascading of frequency-sampling filters to save a factor of R delays for multirate signal processing [4, Sect. 3.4]

filter. An FSF can also be created by cascading all-pole filter sections with all-zero filter (comb) sections, as suggested in Fig. 5.32. The delay of the comb section, $1 \pm z^{-D}$, is chosen so that its zeros cancel the poles of the all-pole prefilter as shown in Fig. 5.33. Wherever there is a complex pole, there is also an annihilating complex zero that results in an all-zero FIR, with the usual linear-phase and constant group-delay properties.

Frequency-sampling filters are of interest to designers of multirate filter banks due, in part, to their intrinsic low complexity and linear-phase behavior. FSF designs rely on exact pole-zero annihilation and are often found in embedded applications. Exact FSF pole-zero annihilation, can be guaranteed by using polynomial filters defined over an *integer ring* using the two's complement or the residue number system (RNS). The poles of an FSF filter developed in this manner can reside on the periphery of the unit circle. This conditionally unstable location is acceptable, due to the guarantee of exact pole-zero cancellation. Without this guarantee, the designer would have to locate the poles of the resonators within the unit circle, with a loss in performance. In addition, by allowing the FSF poles and zeros to reside on the unit circle, a multiplier-less FSF can be created, with an attendant reduction in complexity and an increase in data bandwidth.

Consider the filter shown in Fig. 5.32. It can be shown that first order filter sections (with integer coefficients) produce poles at angles of 0° and 180° . Second order sections, with integer coefficients, can produce poles at angles of 60° , 90° , and 120° , according to the relationship $2\cos(2\pi K/D)=1$, 0 , and -1 . The frequency selectivity of higher order sections is shown in Table 5.4. The angular frequencies for all polynomials having integer coefficients with roots on the unit circle, up to order six, are reported. The building blocks listed in Table 5.4 can be used to efficiently design and implement such FSF filters. For example, a two's complement (i.e., RNS single modulus) filter bank was developed for use as a constant- Q speech processing filter bank. It covers a frequency range from 900 to 8000 Hz [133, 134], using 16 kHz sam-

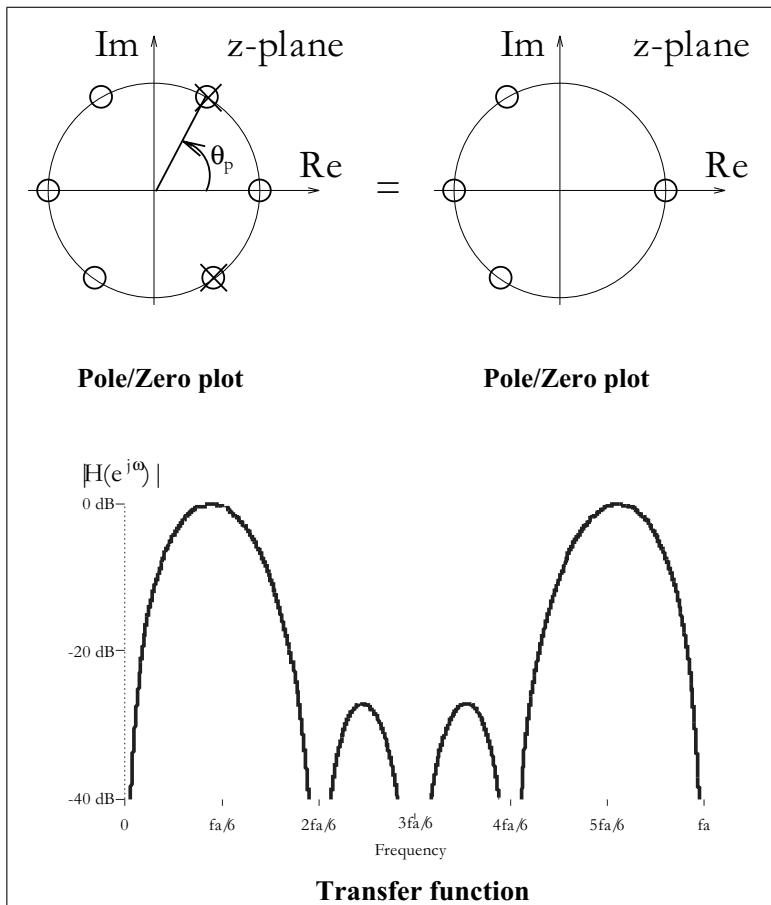


Fig. 5.33. Example of pole/zero-compensation for a pole angle of 60° and comb delay $D = 6$

pling frequency. An integer coefficient half-band filter HB6 [89] anti-aliasing filter and a third order multiplier-free CIC filter (also known as Hogenauer filter [127] see Sect. 5.3, p. 317), was then added to the design to suppress unwanted frequency components, as shown in Fig. 5.34. The bandwidth of each resonator can be independently tuned by the number of stages and delays in the comb section. The number of stages and delays is optimized to meet the desired bandwidth requirements. All frequency-selective filters have two stages and delays.

The filter bank was prototyped using a Xilinx XC4000 FPGA with the complexity reported in Table 5.5. Using high-level design tools (XBLOCKS from Xilinx), the number of used CLBs was typically 20% higher than the

Table 5.4. Filters with integer coefficients producing unique angular pole locations up to order six. Shown are the filter coefficients and nonredundant angular locations of the roots on the unit circle

$C_k(z)$	Order	a_0	a_1	a_2	a_3	a_4	a_5	a_6	θ_1	θ_2	θ_3
$-C_1(z)$	1	1	-1						0°		
$C_2(z)$	1	1	1						180°		
$C_6(z)$	2	1	-1	1					60°		
$C_4(z)$	2	1	0	1					90°		
$C_3(z)$	2	1	1	1					120°		
$C_{12}(z)$	4	1	0	-1	0	1			30°	150°	
$C_{10}(z)$	4	1	-1	1	-1	1			36°	108°	
$C_8(z)$	4	1	0	0	0	1			45°	135°	
$C_5(z)$	4	1	1	1	1	1			72°	144°	
$C_{16}(z)$	6	1	0	0	-1	0	0	1	20.00°	100.00°	140.00°
$C_{14}(z)$	6	1	-1	1	-1	1	-1	1	25.71°	77.14°	128.57°
$C_7(z)$	6	1	1	1	1	1	1	1	51.42°	102.86°	154.29°
$C_9(z)$	6	1	0	0	1	0	0	1	40.00°	80.00°	160.00°

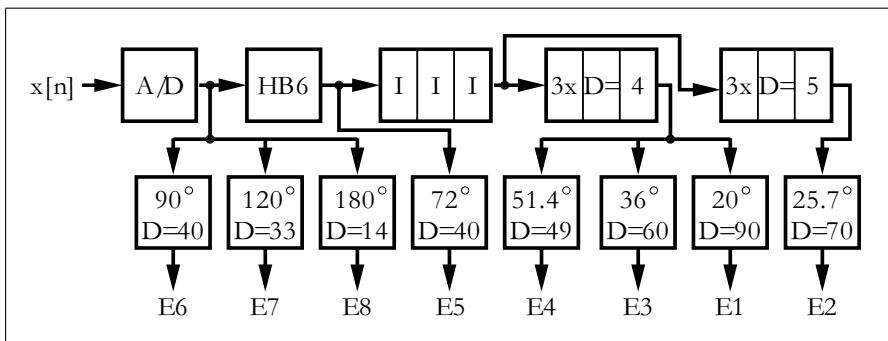


Fig. 5.34. Design of a filter bank consisting of a half-band and CIC prefilter and FSF comb-resonator sections

theoretical prediction obtained by counting adders, flip-flops, ROMs, and RAMs.

The design of an FSF can be manipulated by changing the comb delay, channel amplitude, or the number of sections. For example, adaptation of the comb delay can easily be achieved because the CLBs are used as 32×1 memory cells, and a counter realizes specific comb delays with the CLB used as a memory cell.

Table 5.5. Number of CLBs used in Xilinx XC4000 FPGAs (notation: F20D90 means filter pole angle 20.00°, delay comb $D = 90$). Total: actual 1572 CLBs, nonrecursive FIR: 11292 CLBs

	F20D90	F25D70	F36D60	F51D49	F72D40	F90D40
Theory	122	184	128	164	124	65
Practice	160	271	190	240	190	93
Nonre. FIR	2256	1836	1924	1140	1039	1287
	F120D33	F180D14	HB6	III	D4	D5
Theory	86	35	122	31	24	24
Practice	120	53	153	36	33	33
Nonre. FIR	1260	550				

5.6 Design of Arbitrary Sampling Rate Converters

Most sampling rate converters can be implemented via a rational sampling rate converter system as already discussed. Figure 5.7, p. 309 illustrates the system. Upsampling by R_1 is followed by downsampling R_2 . We now discuss different design options, ranging from IIR, FIR filters, to Lagrange and spline interpolation.

To illustrate, let us look at the design procedure for a rational factor change in the sampling rate with interpolation by $R_1 = 3$ followed by a decimation by $R_2 = 4$, i.e., a rate change by $R = R_1/R_2 = 3/4 = 0.75$.

Example 5.9: $R = 0.75$ Rate Changer I

An interpolation by 3 followed by a decimation by 4 with a centered lowpass filter has to be designed. As shown in Fig. 5.7, p. 309 we only need to implement one lowpass with a cut-off frequency of $\min(\frac{\pi}{3}, \frac{\pi}{4}) = \frac{\pi}{4}$. In MATLAB the frequencies in the filter design procedures are normalized to $f_2/2 = \pi$ and the design of a tenth order Chebyshev II filter with a 50 dB stopband attenuation is accomplished by

```
[B, A] = cheby2(10, 50, 0.25)
```

A Chebyshev II filter was chosen because it has a flat passband, ripple in the stopband, and moderate filter length and is therefore a good choice in many applications. If we want to reduce the filter coefficient sensitivity to quantization, we can use an implementation form with biquad sections rather than the direct forms; see Example 4.2 (p. 237). In MATLAB we use

```
[SOS, gain] = tf2sos(B,A)
```

Using this IIR filter we can now simulate the rational rate change. Figure 5.35 shows a simulation for a triangular test signal. Figure 5.35a shows the original input sequence. (b) the signal after upsampling by 3 and after filtering with the IIR filter, and (c) the signal after downsampling.

5.9

Although the rational interpolation via an IIR filter is not perfect we notice that the triangular shape is well preserved, but ringing of the filter

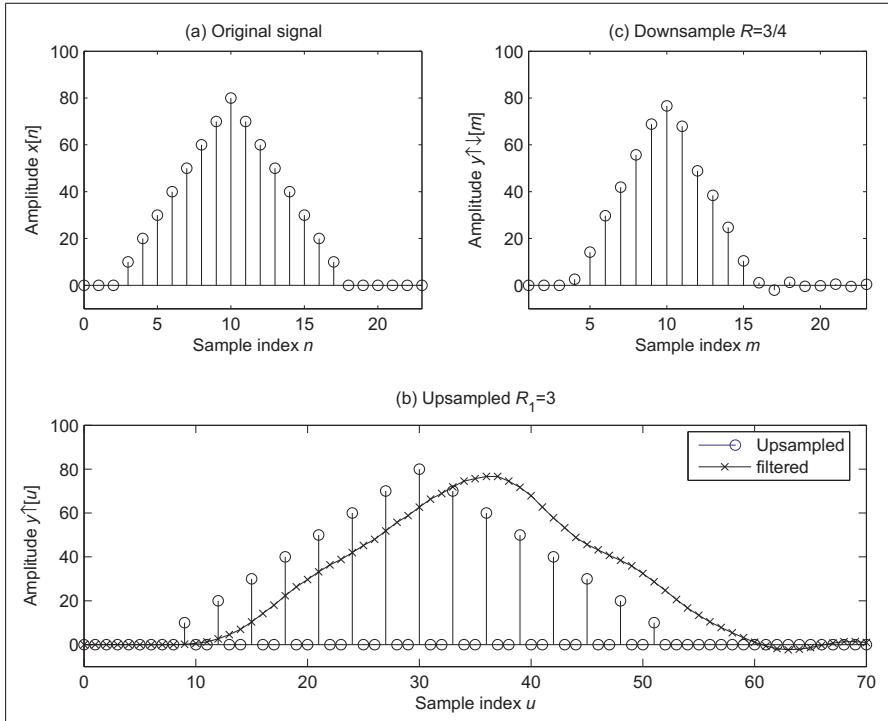


Fig. 5.35. IIR filter simulation of rational rate change. (a) Original signal (b) upsampled and filtered version of the original signal. (c) Signal after downsampling

next to the triangle when the signal should be zero can be observed. We may ask if the interpolation can be improved if we use an exact lowpass that we can build in the frequency domain. Instead of using the IIR filter in the time domain, we may try to interpolate by means of a DFT/FFT-based frequency method [135]. In order to keep the frame processing simple we choose a DFT or FFT whose length N is a multiple of the rate change factors, i.e., $N = k \times R_1 \times R_2$ with $k \in \mathbb{N}$. The necessary processing steps can be summarized as follows:

Algorithm 5.10: Rational Rate Change using an FFT

The algorithm to compute the rate change by $R = R_1/R_2$ via an FFT is as follows:

- 1) Select a block of $k \times R_2$ samples.
- 2) Interpolate with $(R_1 - 1)$ zeros between each sample.
- 3) Compute the FFT of size $N = k \times R_1 \times R_2$.
- 4) Apply the lowpass filter operation in the frequency domain.
- 5) Compute the IFFT of size $N = k \times R_1 \times R_2$.
- 6) Compute finally the output sequence by downsampling by R_1 , i.e., keep $k \times R_2$ samples.

Let us illustrate this algorithm with a small numerical example.

Example 5.11: $R = 0.75$ Rate Changer II

Let us assume we have a triangular input sequence x to interpolate by $R = R_1/R_2 = 3/4 = 0.75$. and we select $k = 1$. The steps are as follows:

- 1) Original block $x = (1, 2, 3, 4)$.
- 2) Interpolation 3 gives $x_i = (1, 0, 0, 2, 0, 0, 3, 0, 0, 4, 0, 0)$.
- 3) The FFT gives $X_i = (10, -2 + j2, -2, 2 - j2, 10, -2 + j2, -2, -2 - j2, 10, -2 + j2, -2, -2 - j2)$.
- 4) The Lowpass filter operation in the frequency domain. $X_{lp} = (10, -2 + j2, -2, 0, 0, 0, 0, 0, 0, -2, -2 - j2)$.
- 5) Compute the IFFT, $y = 3 \times \text{ifft}(X_{lp})$.
- 6) Downsampling finally gives $y = (0.5000, 2.6340, 4.3660)$.

5.11

Let us now apply this block processing to the triangular sequence shown in Fig. 5.36a. From the results shown in Fig. 5.36b we notice the border effects between the blocks when compared with the FFT interpolation results with full-length input data as shown in Fig. 5.36d. This is due to the fact that the underlying assumption of the DFT is that the signals in time and frequency are periodic. We may try to improve the quality and reduce the border disturbance by applying a window function that tapers smoothly to zero at the borders. This will however also reduce the number of useful samples in our output sequence and we need to implement an overlapping block processing. This can be improved by using longer (i.e., $k > 1$) FFTs and removing the leading and tailing samples. Figure 5.36c shows the result for $k = 2$ with removal of 50% of the lead and tail samples. We may therefore ask, why not using a very long FFT, which produces the best results, as can be seen from full length FFT simulation result as shown in Fig. 5.36d? The reason we prefer the short FFT comes from the computational perspective: the longer FFT will require more effort per output sample. Although with $k > 1$ we have more output values per FFT available and overall need fewer FFTs, the computational effort per sample of a radix-2 FFT is $\text{ld}(N)/2$ complex multiplications, because the FFT requires $\text{ld}(N)N/2$ complex multiplications for

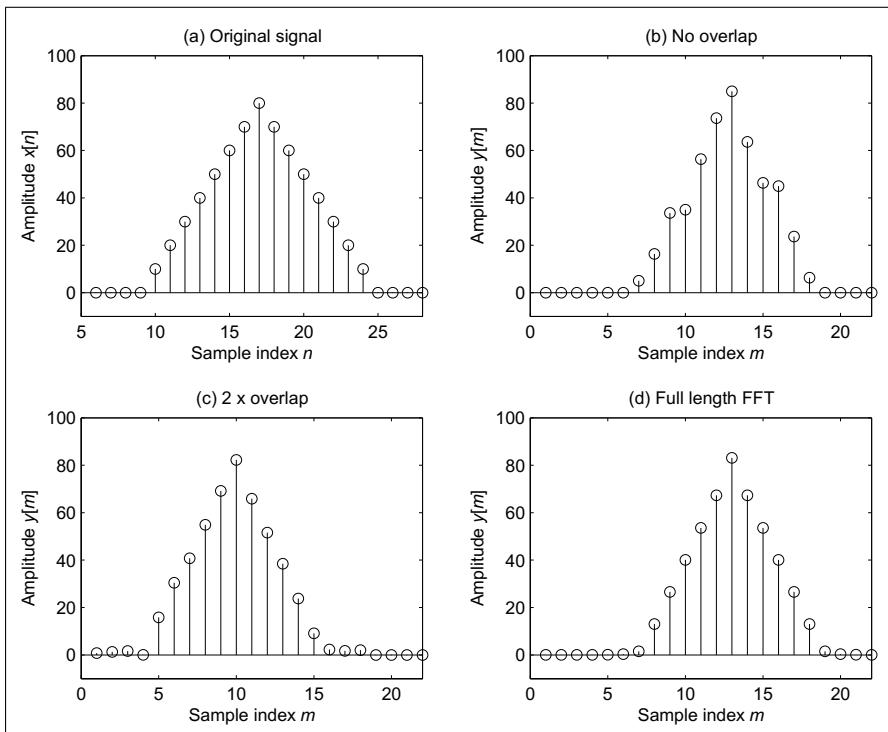


Fig. 5.36. FFT based rational rate change. (a) Original signal (b) Decimation without overlap. (c) 50% overlap. (c) Full length FFT

the N -point radix-2 FFT. A short FFT reduces therefore the computational effort.

Given the contradiction that longer FFTs are used to produce a better approximation while the computational effort requires shorter FFTs, we now want to discuss briefly what computational simplifications can be made to simplify the computation of the two long FFTs in Algorithm 5.10. Two major savings are basically possible.

The first is in the forward transform: the interpolated sequence has many zeros, and if we use a Cooley–Tuckey decimation-in-time-based algorithm, we can group all nonzero values in one DFT block and the remaining $k(R_1 \times R_2 - R_2)$ in the other $R_1 - 1$ groups. Then basically only one FFT of size $k \times R_2$ needs to be computed compared with the full-length $N = k \times R_1 \times R_2$ FFT.

The second simplification can be implemented by computing the downsampling in the frequency domain. For downsampling by two we compute

$$F_{\downarrow 2}(k) = F(k) + F(k + N/2) \quad (5.42)$$

for all $k \leq N/2$. The reason is that the downsampling in the time domain leads to a Nyquist frequency repetition of the base band scaled by a factor of 2. In Algorithm 5.10 we need downsampling by a factor R_2 , which we compute as follows

$$F_{\downarrow R_2}(k) = \sum_n F(k + nN/R_2). \quad (5.43)$$

If we now consider that due to the lowpass filtering in the frequency domain many samples are set to zero, the summations necessary in (5.43) can be further reduced. The IFFT required is only of length $k \times R_1$.

To illustrate the saving let us assume that the implemented FFT and IFFT both require $\text{ld}(N)N/2$ complex multiplications. The modified algorithm is improved by a factor of

$$F = \frac{\frac{2}{2}^{kR_1R_2} \text{ld}(kR_1R_2)}{\frac{kR_1}{2} \text{ld}(kR_1) + \frac{kR_2}{2} \text{ld}(kR_2)}. \quad (5.44)$$

For the simulation above with $R = 3/4$ and 50% overlap we get

$$F = \frac{\text{ld}(24)24}{\text{ld}(6)6/2 + \text{ld}(8)8/2} = 5.57. \quad (5.45)$$

If we can use the Winograd short-term DFT algorithms (see Sect. 6.1.6, p. 434) instead of the Cooley–Tuckey FFT the improvement would be even larger.

5.6.1 Fractional Delay Rate Change

In some applications the input and output sampling rate quotients are close to 1, as in the previous example with $R = 3/4$. As another example consider a change from the DAT player rate (48 kHz) to the CD rate (44.1 kHz), which requires a rational factor change factor of $R = 147/160$. In this case the direct approach using a sampling rate interpolation followed by a sampling rate reduction would required a very high sampling rate for the lowpass interpolation filter. In case of the DAT→CD change for instance the filter must run with 147 times the input sampling rate.

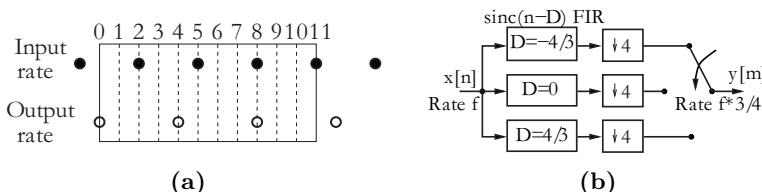


Fig. 5.37. (a) Input and output sampling grid for fractional delay rate change.
(b) Filter configuration for $R = 3/4$ sinc filter system

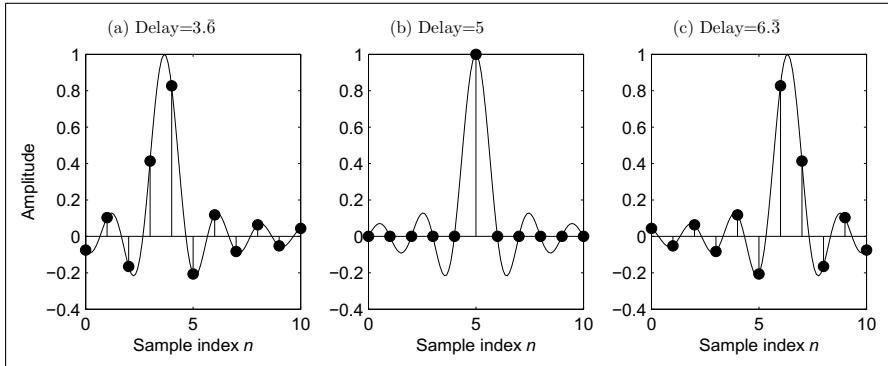


Fig. 5.38. Fractional delay filter with delays $D = 5 - 4/3, 5$, and $5 + 4/3$

In these large- R_1 cases we may consider implementation of the rate change with the help of fractional delays. We will briefly review the idea and then discuss the HDL code for two different versions. The idea will be illustrated by a rate change of $R = 3/4$. Figure 5.37a shows the input and output sampling grid for a system. For each block of four input values, the system computes three interpolated output samples. From a filter perspective we need three filters with unit transfer functions: one filter with a zero delay and two filters that implement the delays of $D = \pm 4/3$. A filter with unit frequency is a sinc or $\sin(t)/t = \text{sinc}(t)$ filter in the time domain. We must allow an initial delay to make the filter realizable, i.e., causal. Figure 5.37b shows the filter configuration and Fig. 5.37 shows the three length-11 filter impulse responses for the delays $5 - 4/3 = 3.\bar{6}$, 5 , and $5 + 4/3 = 6.\bar{3}$.

We can now apply these three filters to our triangular input signal and for each block of four input samples we compute three output samples. Figure 5.39c shows the simulation result for the length-11 sinc filter. Figure 5.39b shows that length-5 filters produce too much ripple to be useful. The length-11 sinc filters produce a much smoother triangular output, but some ripple due to the Gibbs phenomenon can be observed next to the triangular function when the output should be zero.

Let us now have a look at the HDL implementation of the fractional delay rate changer using sinc filters.

Example 5.12: $R = 0.75$ Rate Changer III

The following VHDL code⁷ shows the sinc filter design for an $R = 3/4$ rate change:

```
PACKAGE n_bits_int IS
    -- User defined types
    SUBTYPE U4 IS INTEGER RANGE 0 TO 15;
    SUBTYPE S8 IS INTEGER RANGE -128 TO 127;
    SUBTYPE S9 IS INTEGER RANGE -2**8 TO 2**8-1;
```

⁷ The equivalent Verilog code `rc_sinc.v` for this example can be found in Appendix A on page 835. Synthesis results are shown in Appendix B on page 881.

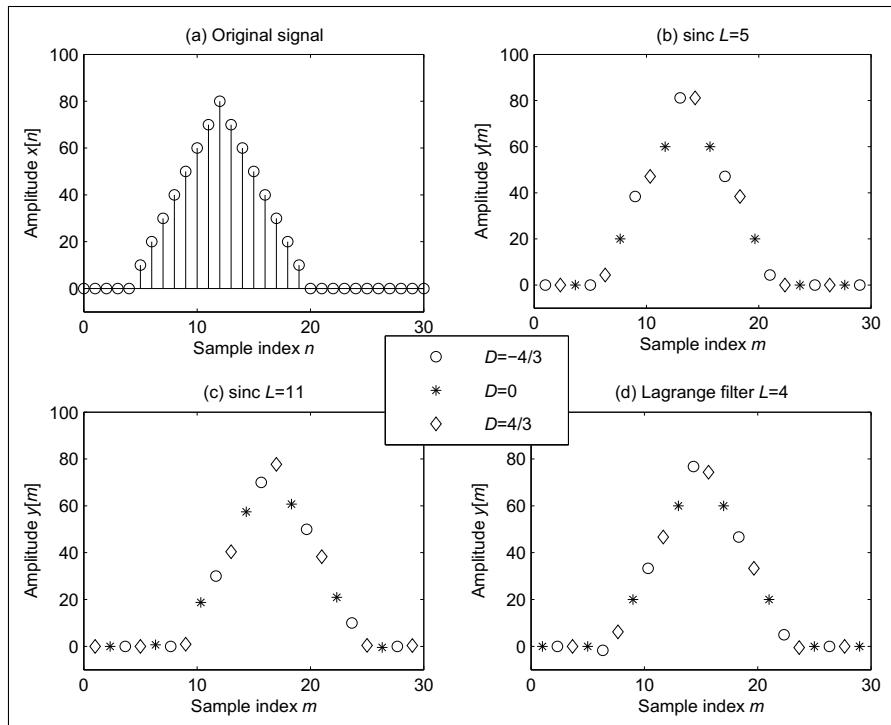


Fig. 5.39. Fraction delay interpolation (a) Original signal (b) Filtering with sinc filter of length 5. (c) Filtering with sinc filter of length 11. (d) Interpolation using Lagrange interpolation

```

SUBTYPE S17 IS INTEGER RANGE -2**16 TO 2**16-1;
TYPE AO_10S8 IS ARRAY (0 TO 10) OF S8;
TYPE AO_10S9 IS ARRAY (0 TO 10) OF S9;
TYPE AO_2S8 IS ARRAY (0 TO 2) OF S8;
TYPE AO_3S8 IS ARRAY (0 TO 3) OF S8;
TYPE AO_10S17 IS ARRAY (0 TO 10) OF S17;
END n_bits_int;

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-----
ENTITY rc_sinc IS                                     -----> Interface
  GENERIC (OL : INTEGER := 2; -- Output buffer length -1
           IL : INTEGER := 3; -- Input buffer length -1
           L  : INTEGER := 10 -- Filter length -1

```

```

        );
PORT (clk      : IN STD_LOGIC; -- System clock
       reset     : IN STD_LOGIC; -- Asynchronous reset
       x_in      : IN S8;   -- System input
       count_o   : OUT U4;   -- Counter FSM
       ena_in_o  : OUT BOOLEAN; -- Sample input enable
       ena_out_o : OUT BOOLEAN; -- Shift output enable
       ena_io_o  : OUT BOOLEAN; -- Enable transfer2output
       f0_o      : OUT S9;   -- First Sinc filter output
       f1_o      : OUT S9;   -- Second Sinc filter output
       f2_o      : OUT S9;   -- Third Sinc filter output
       y_out     : OUT S9); -- System output
END rc_sinc;
-----
ARCHITECTURE fpga OF rc_sinc IS

  SIGNAL count  : U4; -- Cycle R_1*R_2
  SIGNAL ena_in, ena_out, ena_io : BOOLEAN; -- FSM enables
  -- Constant arrays for multiplier and taps:
  CONSTANT c0  : A0_10S9
    := (-19,26,-42,106,212,-53,29,-21,16,-13,11);
  CONSTANT c2  : A0_10S9
    := (11,-13,16,-21,29,-53,212,106,-42,26,-19);
  SIGNAL x : A0_10S8;           -- TAP registers for 3 filters
  SIGNAL ibuf : A0_3S8; -- TAP in registers
  SIGNAL obuf : A0_2S8; -- TAP out registers
  SIGNAL f0, f1, f2 : S9; -- Filter outputs

BEGIN

  FSM: PROCESS (reset, clk)      -----> Control the system
  BEGIN
    IF reset = '1' THEN          -- sample at clk rate
      count <= 0;               -- Asynchronous reset
    ELSIF rising_edge(clk) THEN
      IF count = 11 THEN
        count <= 0;
      ELSE
        count <= count + 1;
      END IF;
    CASE count IS
      WHEN 2 | 5 | 8 | 11 =>
        ena_in <= TRUE;
      WHEN others =>
        ena_in <= FALSE;
    END CASE;
    CASE count IS
      WHEN 4 | 8 =>
        ena_out <= TRUE;
      WHEN others =>
        ena_out <= FALSE;
    END CASE;
  END;

```

```

IF COUNT = 0 THEN
    ena_io <= TRUE;
ELSE
    ena_io <= FALSE;
END IF;
END IF;
END PROCESS FSM;

INPUTMUX: PROCESS(clk, reset) ----> One tapped delay line
BEGIN
    IF reset = '1' THEN -- Asynchronous clear
        FOR I IN 0 TO IL LOOP
            ibuf(I) <= 0;          -- Clear one
        END LOOP;
    ELSIF rising_edge(clk) THEN
        IF ENA_IN THEN
            FOR I IN IL DOWNTO 1 LOOP
                ibuf(I) <= ibuf(I-1);      -- shift one
            END LOOP;
            ibuf(0) <= x_in;           -- Input in register 0
        END IF;
    END IF;
END PROCESS;

OUPUTMUX: PROCESS(clk, reset) ----> One tapped delay line
BEGIN
    IF reset = '1' THEN -- Asynchronous clear
        FOR I IN 0 TO OL LOOP
            obuf(I) <= 0;          -- Clear one
        END LOOP;
    ELSIF rising_edge(clk) THEN
        IF ENA_IO THEN -- store 3 samples in output buffer
            obuf(0) <= f0 ;
            obuf(1) <= f1;
            obuf(2) <= f2 ;
        ELSIF ENA_OUT THEN
            FOR I IN OL DOWNTO 1 LOOP
                obuf(I) <= obuf(I-1);      -- shift one
            END LOOP;
        END IF;
    END IF;
END PROCESS;

TAP: PROCESS(clk, reset)      -----> One tapped delay line
BEGIN
    IF reset = '1' THEN -- Asynchronous clear
        FOR I IN 0 TO 10 LOOP
            x(I) <= 0;          -- Clear register
        END LOOP;
    ELSIF rising_edge(clk) THEN
        IF ENA_IO THEN
            FOR I IN 0 TO 3 LOOP -- take over input buffer
                x(I) <= ibuf(I);
            END LOOP;
        END IF;
    END IF;
END PROCESS;

```

```

        END LOOP;
        FOR I IN 4 TO 10 LOOP -- 0->4; 4->8 etc.
            x(I) <= x(I-4);           -- shift 4 taps
        END LOOP;
    END IF;
END IF;
END PROCESS;

SOP0: PROCESS(clk, reset, x) --> Compute sum-of-products
VARIABLE sum : S17;                                -- for f0
VARIABLE p : AO_10S17;
BEGIN
    FOR I IN 0 TO L LOOP -- Infer L+1 multiplier
        p(I) := c0(I) * x(I);
    END LOOP;
    sum := p(0);
    FOR I IN 1 TO L LOOP      -- Compute the direct
        sum := sum + p(I);      -- filter adds
    END LOOP;
    IF reset = '1' THEN -- Asynchronous clear
        f0 <= 0;
    ELSIF rising_edge(clk) THEN
        f0 <= sum /256;
    END IF;
END PROCESS SOP0;

SOP1: PROCESS(clk, reset) --> Compute sum-of-products
BEGIN                                         -- for f1
    IF reset = '1' THEN -- Asynchronous clear
        f1 <= 0;
    ELSIF rising_edge(clk) THEN
        f1 <= x(5); -- No scaling, i.e. unit impulse
    END IF;
END PROCESS SOP1;

SOP2: PROCESS(clk, reset, x) --> Compute sum-of-products
VARIABLE sum : S17;                                -- for f2
VARIABLE p : AO_10S17;
BEGIN
    FOR I IN 0 TO L LOOP -- Infer L+1 multiplier
        p(I) := c2(I) * x(I);
    END LOOP;
    sum := p(0);
    FOR I IN 1 TO L LOOP      -- Compute the direct
        sum := sum + p(I);      -- filter adds
    END LOOP;
    IF reset = '1' THEN -- Asynchronous clear
        f2 <= 0;
    ELSIF rising_edge(clk) THEN
        f2 <= sum /256;
    END IF;
END PROCESS SOP2;

```

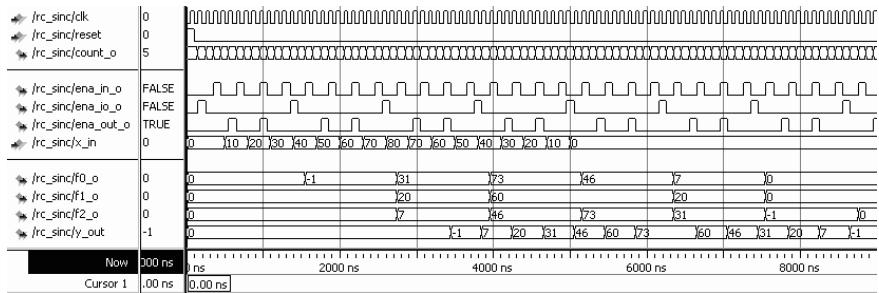


Fig. 5.40. VHDL simulation of the $R = 3/4$ rate change using three sinc filter

```

f0_o <= f0;           -- Provide some test signal as outputs
f1_o <= f1;
f2_o <= f2;
count_o <= count;
ena_in_o <= ena_in;
ena_out_o <= ena_out;
ena_io_o <= ena_io;

y_out <= obuf(OL); -- Connect to output

```

END fpga;

The first PROCESS is the FSM, which includes the control flow and generation of the enable signals for the input and output buffers, and the enable signal **ena_io** for the three filters. The full round takes 12 clock cycles. The next three PROCESS blocks include the input buffer, output buffer, and the TAP delay line. Note that only one tapped delay line is used for all three filters. The final three PROCESS blocks include the sinc filter. The output **y_out** was chosen to have an additional guard bit. The design uses 880 LEs, no embedded multiplier, and has an **Fmax**=59.53 MHz registered performance using the TimeQuest slow 85C model.

A simulation of the filter is shown in Fig. 5.40. The simulation first shows the control and enable signals of the FSM. A triangular input **x_in** is used. The three filter outputs only update once every 12 clock cycles. The filter output values (**f0**, **f1**, **f2**) are arranged in the correct order to generate the output **y_out**. Note that the filter values 20 and 60 from **f1** appear unchanged in the output sequence, while the other values are interpolated.

5.12

Notice that in this particular case the filters are implemented in the direct rather than in the transposed form, because now we only need one tapped delay line for all three filters. Due to the complexity of the design the coding style for this example was based more on clarity than efficiency. The filters can be improved if MAG coding is used for the filter coefficients and a pipelined adder tree is applied at the filter summations; see Exercise 5.15 (p. 413).

5.6.2 Polynomial Fractional Delay Design

The implementation of the fractional delay via a set of lowpass filters is attractive as long as the number of delays (i.e., nominator R_1 in the rate change factor $R = R_1/R_2$ to be implemented) is small. For large R_1 however, as for examples required for the DAT→CD conversion with rate change factor $R = 147/160$, this implies a large implementation effort, because 147 different lowpass filters need to be implemented. It is then more attractive to compute the fractional delay via a polynomial approximation using Lagrange or spline polynomials [136, 137]. An N -point so-called Lagrange polynomial approximation will be of the type:

$$p(t) = c_0 + c_1 t + c_2 t^2 + \dots + c_{N-1} t^{N-1}, \quad (5.46)$$

where typically 3-4 points should be enough, although some high-quality audio application use up to 10 terms [138]. The Lagrange polynomial approximation has the tendency to oscillate at the ends, and the interval to be estimated should be at the center of the polynomial. Figure 5.41 illustrates this fact for a signal with just two nonzero values. It can also be observed that a length-4 polynomial already gives a good approximation for the center interval, i.e., $0 \leq n \leq 1$, and that the improvement from a length-4 to a length-8 polynomial is not significant. A bad choice of the approximation interval would be the first or last interval, e.g., the range $3 \leq n \leq 4$ for a length-8 polynomial. This choice would show large oscillations and errors. The input sample set in use should therefore be placed symmetric around the interval for which the fractional delay should be approximated such that $0 \leq d \leq 1$. We use input samples at times $-N/2-1, \dots, N/2$. For four points for example we will use input samples at $-1, 0, 1, 2$. In order to fit the polynomial $p(t)$ through the sample points we substitute the sample times and $x(t)$ values into (5.46) and solve this equation for the coefficients c_k . This matrix equation $\mathbf{V}\mathbf{c} = \mathbf{x}$ leads to a so-called Lagrange polynomial [86, 136] and for $N = 4$, for instance, we get:

$$\begin{bmatrix} 1 & t_{-1} & t_{-1}^2 & t_{-1}^3 \\ 1 & t_0 & t_0^2 & t_0^3 \\ 1 & t_1 & t_1^2 & t_1^3 \\ 1 & t_2 & t_2^2 & t_2^3 \end{bmatrix} \times \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} x(n-1) \\ x(n) \\ x(n+1) \\ x(n+2) \end{bmatrix} \quad (5.47)$$

with $t_k = k$; we need to solve this equation for the unknown coefficients c_n . We also notice that the matrix for the t_k is a Vandermonde matrix a popular matrix type we also use for the DFT. Each line in the Vandermonde matrix is constructed by building the power series of a basic element, i.e., $t_k^l = 1, t_k, t_k^2, \dots$. Substitution of the t_k and matrix inversion leads to

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \end{bmatrix}^{-1} \times \begin{bmatrix} x(n-1) \\ x(n) \\ x(n+1) \\ x(n+2) \end{bmatrix}$$

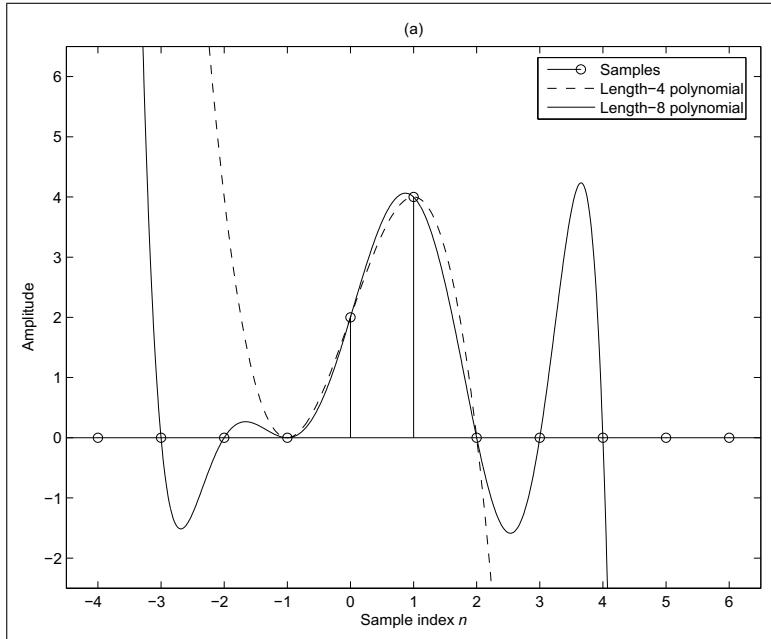


Fig. 5.41. Polynomial approximation using a short and long polynomial

$$= \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\frac{1}{3} & -\frac{1}{2} & 1 & -\frac{1}{6} \\ \frac{1}{2} & -1 & \frac{1}{2} & 0 \\ -\frac{1}{6} & \frac{1}{2} & -\frac{1}{2} & \frac{1}{6} \end{bmatrix} \times \begin{bmatrix} x(n-1) \\ x(n) \\ x(n+1) \\ x(n+2) \end{bmatrix} \quad (5.48)$$

For each output sample we now need to determine the fractional delay value d , solve the matrix equation (5.48), and finally compute the polynomial approximation via (5.46). A simulation using the Lagrange approximation is shown in Fig. 5.39d, p. 351. This give a reasonably exact approximation, with little or no ripple in the Lagrange approximation, compared to the sinc design next to the triangular values where the input and output values are supposed to be zero.

The polynomial evaluation (5.46) can be more efficiently computed if we use the Horner scheme instead of the direct evaluation of (5.46). Instead of

$$p(d) = c_0 + c_1 d + c_2 d^2 + c_3 d^3 \quad (5.49)$$

we use for $N = 4$ the Horner scheme

$$p(d) = c_0 + d(c_1 + d(c_2 + d(c_3))). \quad (5.50)$$

The advantage is that we do not need to evaluate the power of d^k values. This was first suggested by Farrow [139] and is therefore called in the literature the Farrow structure [140–143]. The Farrow structure for four coefficients is shown in Fig. 5.42b.

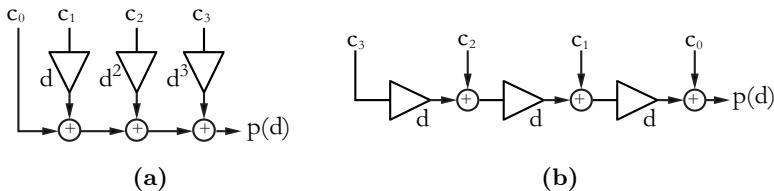


Fig. 5.42. Fractional delay via an $N = 4$ polynomial approximation. (a) Direct implementation. (b) Farrow structure

Let us now look at an implementation example of the polynomial fractional delay design.

Example 5.13: $R = 0.75$ Rate Changer IV

The following VHDL code⁸ shows the Farrow design using a Lagrange polynomial of order 3 for a $R = 3/4$ rate change:

```

PACKAGE n_bits_int IS          -- User defined types
    SUBTYPE U4 IS INTEGER RANGE 0 TO 15;
    SUBTYPE S8 IS INTEGER RANGE -128 TO 127;
    SUBTYPE S9 IS INTEGER RANGE -2**8 TO 2**8-1;
    SUBTYPE S17 IS INTEGER RANGE -2**16 TO 2**16-1;
    TYPE A0_3S8 IS ARRAY (0 TO 3) OF S8;
END n_bits_int;

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-----
ENTITY farrow IS           -----> Interface
    GENERIC (IL : INTEGER := 3); -- Input buffer length -1
    PORT (clk           : IN STD_LOGIC; -- System clock
          reset         : IN STD_LOGIC; -- Asynchronous reset
          x_in          : IN S8;        -- System input
          count_o       : OUT U4;      -- Counter FSM
          ena_in_o     : OUT BOOLEAN; -- Sample input enable
          ena_out_o    : OUT BOOLEAN; -- Shift output enable
          c0_o, c1_o, c2_o, c3_o : OUT S9; -- Phase delays
          d_out         : OUT S9;      -- Delay used
          y_out         : OUT S9);    -- System output
END farrow;
-----
ARCHITECTURE fpga OF farrow IS

```

⁸ The equivalent Verilog code `farrow.v` for this example can be found in Appendix A on page 839. Synthesis results are shown in Appendix B on page 881.

```

SIGNAL count : INTEGER RANGE 0 TO 12; -- Cycle R_1*R_2
CONSTANT delta : INTEGER := 85; -- Increment d
SIGNAL ena_in, ena_out : BOOLEAN; -- FSM enables
SIGNAL x, ibuf : A0_3S8 := (0,0,0,0); -- TAP reg.
SIGNAL d : S9 := 0; -- Fractional Delay scaled to 8 bits
-- Lagrange matrix outputs:
SIGNAL c0, c1, c2, c3 : S9 := 0;

BEGIN

  FSM: PROCESS (reset, clk)      -----> Control the system
  VARIABLE dnew : S9 := 0;
  BEGIN
    -- sample at clk rate
    IF reset = '1' THEN           -- Asynchronous reset
      count <= 0;
      d <= delta;
    ELSIF rising_edge(clk) THEN
      IF count = 11 THEN
        count <= 0;
      ELSE
        count <= count + 1;
      END IF;
      CASE count IS
        WHEN 2 | 5 | 8 | 11 =>
          ena_in <= TRUE;
        WHEN others =>
          ena_in <= FALSE;
      END CASE;
      CASE count IS
        WHEN 3 | 7 | 11 =>
          ena_out <= TRUE;
        WHEN others =>
          ena_out <= FALSE;
      END CASE;
    -- Compute phase delay
    IF ENA_OUT THEN
      dnew := d + delta;
      IF dnew >= 255 THEN
        d <= 0;
      ELSE
        d <= dnew;
      END IF;
    END IF;
  END PROCESS FSM;

  TAP: PROCESS(clk, reset)      -----> One tapped delay line
  BEGIN
    IF reset = '1' THEN -- Asynchronous clear
      FOR I IN 0 TO IL LOOP
        ibuf(I) <= 0;      -- Clear register
      END LOOP;
    ELSIF rising_edge(clk) THEN

```

```

        IF ENA_IN THEN
            FOR I IN 1 TO IL LOOP
                ibuf(I-1) <= ibuf(I);           -- Shift one
            END LOOP;
            ibuf(IL) <= x_in;             -- Input in register IL
        END IF;
    END IF;
END PROCESS;

GET: PROCESS(clk, reset) -----> Get 4 samples at one time
BEGIN
    IF reset = '1' THEN -- Asynchronous clear
        FOR I IN 0 TO IL LOOP
            x(I) <= 0;                  -- Clear register
        END LOOP;
    ELSIF rising_edge(clk) THEN
        IF ENA_OUT THEN
            FOR I IN 0 TO IL LOOP -- take over input buffer
                x(I) <= ibuf(I);
            END LOOP;
        END IF;
    END IF;
END PROCESS;

--> Compute sum-of-products:
SOP: PROCESS (clk, reset, ENA_OUT, c0, c1, c2, c3, d)
VARIABLE y : S9;
BEGIN
-- Matrix multiplier iV=inv(Vandermonde) c=iV*x(n-1:n+2)'
--      x(0)      x(1)      x(2)      x(3)
-- iV=   0       1.0000      0       0
--      -0.3333   -0.5000   1.0000  -0.1667
--      0.5000   -1.0000   0.5000      0
--     -0.1667    0.5000   -0.5000   0.1667
    IF reset = '1' THEN                      -- Asynchronous clear
        y_out <= 0;
        c0 <= 0; c1 <= 0; c2<= 0; c3 <= 0;
    ELSIF ENA_OUT THEN
        IF rising_edge(clk) THEN
            c0 <= x(1);
            c1 <= -85 * x(0)/256 - x(1)/2 + x(2) - 43 * x(3)/256;
            c2 <= (x(0) + x(2)) /2 - x(1) ;
            c3 <= (x(1) - x(2))/2 + 43 * (x(3) - x(0))/256;
        END IF;
    -- Farrow structure = Lagrange with Horner schema
    -- for u=0:3, y=y+f(u)*d^u; end;
    y := c2 + (c3 * d) / 256; -- d is scale by 256
    y := (y * d) / 256 + c1;
    y := (y * d) / 256 + c0;

    IF rising_edge(clk) THEN
        y_out <= y; -- Connect to output + store in register

```

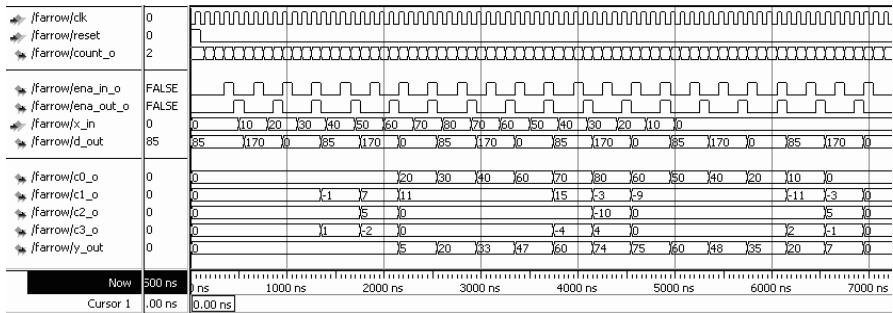


Fig. 5.43. VHDL simulation of the $R = 3/4$ rate change using Lagrange polynomials and a Farrow combiner

```

        END IF;
END IF;
END PROCESS SOP;

c0_o <= c0;      -- Provide some test signals as outputs
c1_o <= c1;
c2_o <= c2;
c3_o <= c3;
count_o <= count;
ena_in_o <= ena_in;
ena_out_o <= ena_out;
d_out <= d;

END fpga;

```

The HDL code for the control is similar to the `rc_sinc` design discussed in Example 5.9 (p. 345). The first `PROCESS` is the FSM, which includes the control flow and generation of the enable signals for input, output buffer, and the computation of the delay D . The full round takes 12 clock cycles. The next two `PROCESS` blocks include the input buffer and the TAP delay line. Note that only one tapped delay line is used for all four polynomial coefficients c_k . The SOP `PROCESS` blocks includes the Lagrange matrix computation and the Farrow combiner. The output `y_out` was chosen to have an additional guard bit. The design uses 363 LEs, three embedded multipliers, and has an `Fmax`=39.82 MHz registered performance using the `TimeQuest` slow 85C model.

A simulation of the filter is shown in Fig. 5.43. The simulation shows first the control and enable signals of the FSM. A triangular input `x_in` is used. The three filter outputs only update once every four clock cycles, i.e., three times in an overall cycle. The filter output values are weighted using the Farrow structure to generate the output `y_out`. Note that only the first and second Lagrange polynomial coefficients are nonzero, due to the fact that a triangular input signal does not have higher polynomial coefficient. Notice also that the filter values 20 and 60 from `c0` appear unchanged in the output sequence (because $D = 0$ at these points in time), while the other values are interpolated.

Although the implementation data for the Lagrange interpolation with the Farrow combiner and the sinc filter design do not differ much for our example design with $R = R_1/R_2 = 3/4$, larger differences occur when we try to implement rate changes with large values of R_1 . The discussed Farrow design only needs to be changed in the enable signal generation. The effort for the Lagrange interpolation and Farrow combiner remain the same, while for a sinc filter the design effort will be proportional to the number of filters to be implemented, i.e., R_1 ; see Exercise 5.16 (p. 413). The only disadvantage of the Farrow combiner is the long latency due to the sequential organization of the multiplications, but this can be improved by adding pipeline stages for the multipliers and coefficient data; see Exercise 5.17 (p. 413).

5.6.3 B-Spline-Based Fractional Rate Changer

Polynomial approximation using Lagrange polynomials is smooth in the center but has the tendency to have large ripples at the end of the polynomials; see Fig. 5.41, p. 357. Much smoother behavior is promised when B-spline approximation functions are used. In contrast to Lagrange polynomials B-splines are of finite length and a B-spline of degree N must be N -times differentiable, hence the smooth behavior. Depending on the border definitions, several versions of B-splines can be defined [86, p.113-116], but the most popular are those defined via the integration of the box function, as shown in Fig. 5.44. A B-spline of degree zero is integrated to give a triangular B-spline, degree-one B-spline integrated yields a quadratic function, etc.

An analytic description of a B-spline is possible [144, 145] using the following representation of the ramp function:

$$(t - \tau)_+ = \begin{cases} t - \tau & \forall t > \tau \\ 0 & \text{otherwise} \end{cases}. \quad (5.51)$$

This allows us to represent the N^{th} -degree symmetric B-spline as

$$\beta^N(t) = \frac{1}{N!} \sum_{k=0}^{N+1} (-1)^k \binom{N+1}{k} \left(t - k + \frac{N+1}{2} \right)_+^N. \quad (5.52)$$

All segments of the B-splines use polynomials of degree N and are therefore N -times differentiable, resulting in a smooth behavior also at the end of the B-splines. Zero- and first-degree B-splines give box and triangular representations, respectively; quadratic and cubic B-splines are next. Cubic B-splines are the most popular type used in DSP although for very high-quality speech processing degree six has been used [138]. For a cubic B-spline (5.52) for instance we get

$$\begin{aligned} \beta^3(t) &= \frac{1}{6} \sum_{k=0}^4 (-1)^k \binom{4}{k} (t - k + 2)_+^3 \\ &= \frac{1}{6}(t+2)_+^3 - \frac{2}{3}(t+1)_+^3 + t_+^3 - \frac{2}{3}(t-1)_+^3 + \frac{1}{6}(t-2)_+^3 \end{aligned}$$

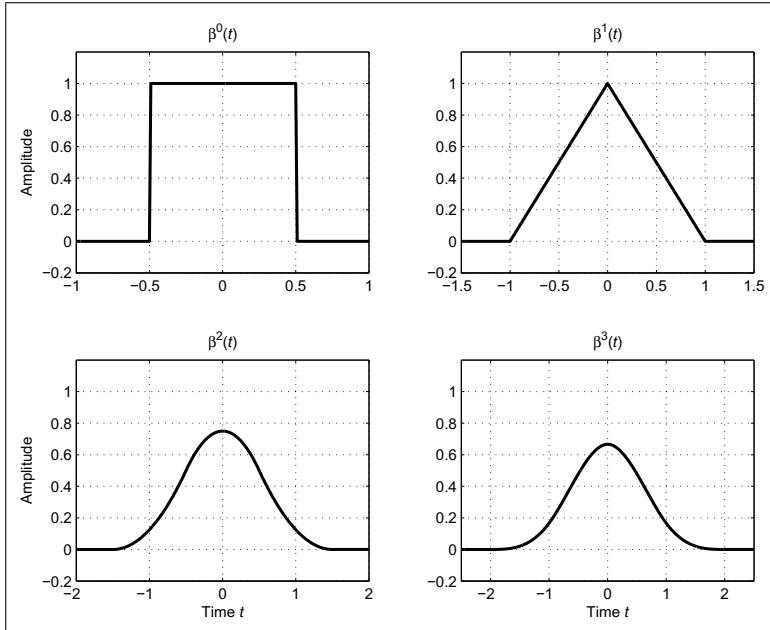


Fig. 5.44. B-spline functions of degree zero to three

$$= \underbrace{\frac{1}{6}(t+2)^3}_{t>-2} - \underbrace{\frac{2}{3}(t+1)^3}_{t>-1} + \underbrace{t^3}_{t>0} - \underbrace{\frac{2}{3}(t-1)^3}_{t>1} + \underbrace{\frac{1}{6}(t-2)^3}_{t>2}. \quad (5.53)$$

We can now use this cubic B-spline for the reconstruction of the spline, by summation of the weighted sequence of the B-splines, i.e.,

$$\hat{y}(t) = \sum_k x(k) \beta^3(t-k). \quad (5.54)$$

This weighted sum is shown in Fig. 5.45 as a bold line. Although $\hat{y}(t)$ is quite smooth, we can also observe that the spline $\hat{y}(t)$ does not go exactly through the sample points, i.e., $\hat{y}(k) \neq x(k)$. Such a B-spline reconstruction is called in the literature a B-spline *approximation*. From a B-spline *interpolation*, however, we expect that the weighted sum goes exactly through our sample points [146]. For cubic B-splines for instance it turns out [141, 147] that the cubic B-spline applies a filter weight whose z -transform is given by

$$H(z) = \frac{z+4+z^{-1}}{6} \quad (5.55)$$

to the sample points. To achieve a perfect interpolation we therefore need to apply an inverse cubic B-spline filter, i.e.,

$$F(z) = 1/H(z) = \frac{6}{z+4+z^{-1}} \quad (5.56)$$

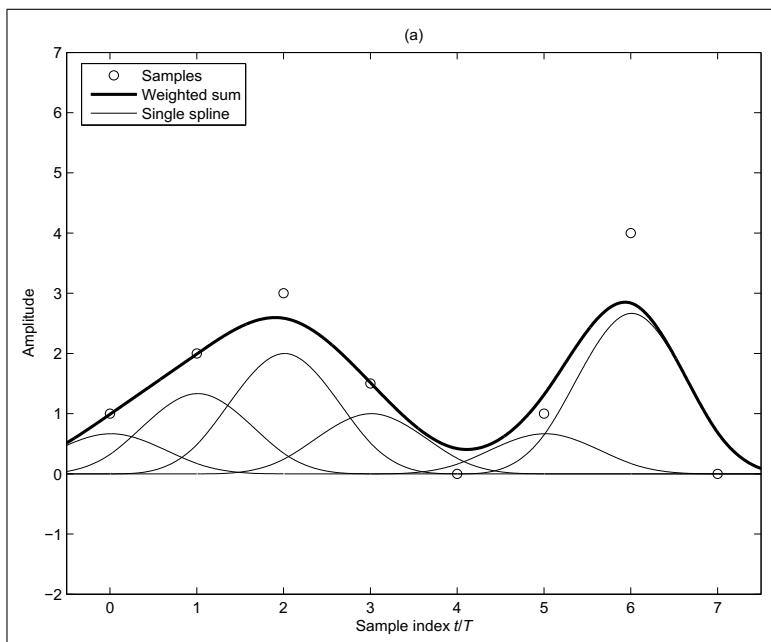


Fig. 5.45. Spline approximation using cubic B-splines

to our input samples. Unfortunately the pole/zero plot of this filter reveals that this IIR filter is not stable and, if we apply this filter to our input sequence, we may produce an increasing signal for the impulse response; see Exercise 5.18 (p. 414). Unser et al. [148] suggest splitting the filter into a stable, causal part and a stable, a-causal filter part and applying the a-causal filter starting with the last value of the output of the first causal filter. While this works well in image processing with a finite number of samples in each image line, in a continuous signal processing scheme this is not practical, especially when the filters are implemented with finite arithmetic.

However another approach that can be used for continuous signal processing is to approximate the filter $F(z) = 1/H(z)$ by an FIR filter. It turns out that even very few FIR coefficients give a good approximation, because the transfer function does not have any sharp edges; see Fig. 5.46. We just need to compute the transfer function of the IIR filter and then take the IFFT of the transfer function to determine the FIR time values. We may also apply a bias correction if a DC shift is critical in the application. Unser et al. [149] suggested an algorithm to optimize the filter coefficient set, but due to the nature of the finite coefficient precision and finite coefficient set, the gain compared with the direct IFFT method is not significant; see Fig. 5.46 and Exercise 5.19, p. 414.

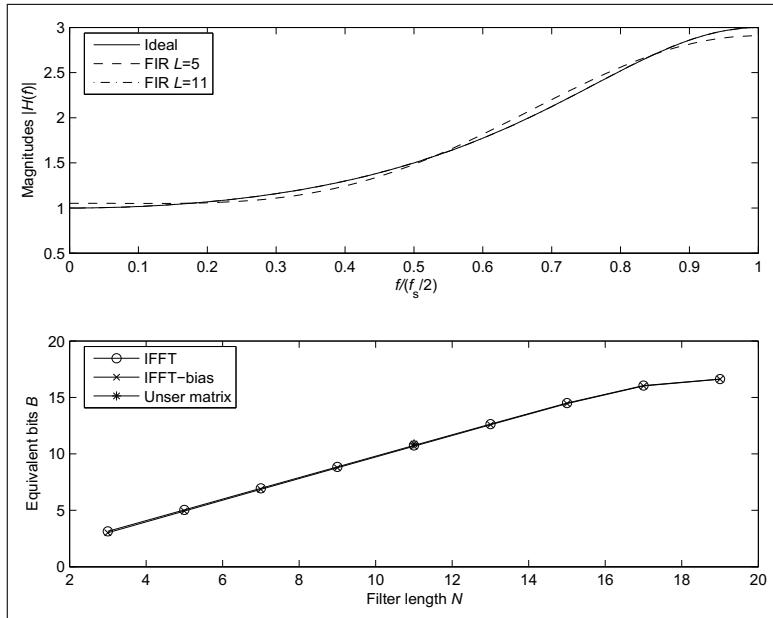


Fig. 5.46. FIR compensation filter design for cubic B-spline interpolation

Now we can apply this FIR filter first to our input samples and then use a cubic B-spline reconstruction. As can be seen from Fig. 5.47 we have in fact an interpolation, i.e., the reconstructed function goes through our original sampling points, i.e., $\hat{y}(k) = x(k)$.

The only thing left to do is to develop a fractional delay B-spline interpolation and to determine the Farrow filter structure. We want to use the popular cubic B-spline set and only consider fractional delays in the range $0 \leq d \leq 1$. For an interpolation with four points we use the samples at time instances $t = -1, 0, 1, 2$ of the input signal [140, p. 780]. With the B-spline representation (5.53) and the weighted sum (5.54) we also find that four B-spline segments have to be considered and we arrive at

$$\begin{aligned} y(d) &= x(n+2)\beta^3(d-2) + x(n+1)\beta^3(d-1) \\ &\quad + x(n)\beta^3(d) + x(n-1)\beta^3(d-1) \end{aligned} \tag{5.57}$$

$$\begin{aligned} &= x(n+2)\frac{d^3}{6} + x(n+1)\left[\frac{1}{6}(d+1)^3 - \frac{2}{3}d^3\right] + x(n) \times \\ &\quad \left[d^3 - \frac{2}{3}(d+1)^3 + \frac{1}{6}(d+2)^3\right] + x(n-1)\left[-\frac{1}{6}(d-1)^3\right] \end{aligned} \tag{5.58}$$

$$= x(n+2)\frac{d^3}{6} + x(n+1)\left[-\frac{d^3}{2} + \frac{d^2}{2} + \frac{d}{2} + \frac{1}{6}\right]$$

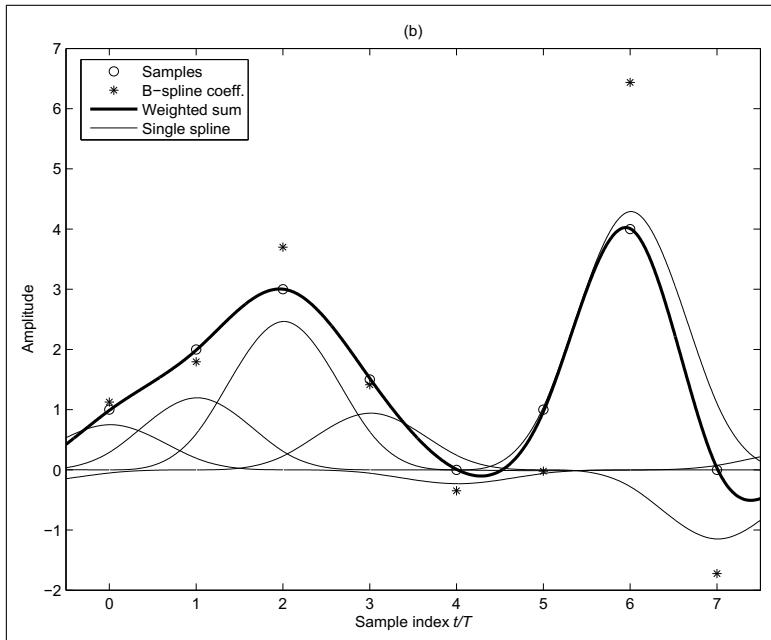


Fig. 5.47. Interpolation using cubic B-splines and FIR compensation filter

$$+x(n) \left[\frac{d^3}{2} - d^2 + \frac{2}{3} \right] + x(n-1) \left[-\frac{d^3}{6} + \frac{d^2}{2} - \frac{d}{2} + \frac{1}{6} \right]. \quad (5.59)$$

In order to realize this in a Farrow structure we need to summarize according to the factors d^k , which yields the following four equations:

$$\begin{aligned} d^0 : \quad 0 & \quad +x(n+1)/6 + 2x(n)/3 + x(n-1)/6 = c_0 \\ d^1 : \quad 0 & \quad +x(n+1)/2 \quad +0 \quad -x(n-1)/2 = c_1 \\ d^2 : \quad 0 & \quad +x(n+1)/2 \quad -x(n) \quad +x(n-1)/2 = c_2 \\ d^3 : x(n+2)/6 & \quad -x(n+1)/2 \quad x(n)/2 \quad -x(n-1)/6 = c_3 \end{aligned} \quad (5.60)$$

This Farrow structure can be translated directly into a B-spline rate changer as discussed in Exercise 5.23, p. 415.

5.6.4 MOMS Fractional Rate Changer

One aspect⁹ often overlooked in traditional design of interpolation kernels $\phi(t)$ is the *order* of the approximation, which is an essential parameter in the quality of the interpolation result. Here the order is defined by the rate of decrease of the square error (i.e., L^2 norm) between the original function and the reconstructed function when the sampling step vanishes. In terms

⁹ This section was suggested by P. Thévenaz from EPFL.

of implementation effort the *support* or length of the interpolation function is a critical design parameter. It is now important to notice that the B-splines used in the last section is both maximum order and minimum support (MOMS) [150]. The question then is whether or not the B-spline is the only kernel that has degree $L - 1$, support of length L , and order L . It turns out that there is a whole class of functions that obey this MOMS behavior. This class of interpolating polynomials can be described as

$$\phi(t) = \beta^N(t) + \sum_{k=1}^N p(k) \frac{d^k \beta^N(t)}{dt^k}. \quad (5.61)$$

Since B-splines are built via successive convolution with the box function the differentiation can be computed via

$$\frac{d\beta^{k+1}(t)}{dt} = \beta^k \left(t + \frac{1}{2} \right) - \beta^k \left(t - \frac{1}{2} \right). \quad (5.62)$$

From (5.61) it can be seen that we have a set of design parameters $p(k)$ at hand that can be chosen to meet certain design goals. In many designs symmetry of the interpolation kernel is desired, forcing all odd coefficients $p(k)$ to zero. A popular choice is $N = 3$, i.e., the cubic spline type, and it follows then that:

$$\begin{aligned} \phi(t) &= \beta^3(t) + p(2) \frac{d^2 \beta^3(t)}{dt^2} \\ &= \beta^3(t) + p(2) (\beta^1(t+1) - 2\beta^1(t) + \beta^1(t-1)) \end{aligned} \quad (5.63)$$

and only the design parameter $p(2)$ needs to be determined. We may, for instance, try to design a direct interpolating function that requires no compensation filter at all. Those I-MOMS occur for $p(2) = -1/6$, and are identical to the Lagrange interpolation (see Exercise 5.20, p. 414) and therefore give sub-optimal interpolation results. Figure 5.48b shows the I-MOMS interpolation of degree three. Another design goal may be to minimize the interpolation error in the L^2 norm sense. These O-MOMS require $p(2) = 1/42$, and the approximation error is a magnitude smaller than for I-MOMS [151]. Figure 5.48c shows the O-MOMS interpolation kernel for degree three. We may also use an iterative method to maximize a specific application the S/N of the interpolation. For a specific set of five images, for instance, $p(2) = 1/28$ has been found to perform one dB better than O-MOMS [152].

Unfortunately the compensation filter required for O-MOMS has (as for B-splines) an unstable pole location and an FIR approximation has to be used in a continuous signal processing scheme. The promised gain via the small L^2 error of the O-MOMS will therefore most likely not result in much overall gain if the FIR has to be built in finite-precision arithmetic. If we give up the symmetry requirements of the kernel then we can design MOMS functions in such a way that the interpolation function sampled at integer points $\phi(k)$ is a causal function, i.e., $\phi(-1) = 0$, as can be seen from Fig. 5.48d. This

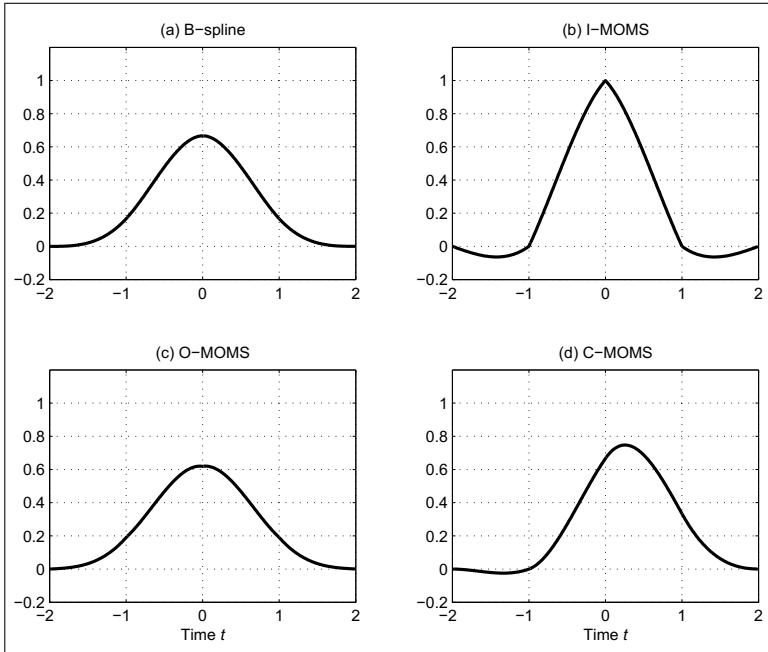


Fig. 5.48. Possible MOMS kernel functions of length four

C-MOMS function is fairly smooth since $p(2) = p(3) = 0$. The C-MOMS requirement demands $p(1) = -1/3$ and we get the asymmetric interpolation function, but with the major advantage that a simple one-pole stable IIR compensation filter with $F(z) = 1.5/(1 + 0.5z^{-1})$ can be used. No FIR approximation is necessary as for B-splines or O-MOMS [151]. It is now interesting to observe that the C-MOMS maxima and sample points no longer have the same time location as in the symmetric kernel, e.g., B-spline case. To see this compare Fig. 5.47 with Fig. 5.49. However, the weighted sum of the C-MOMS goes thorough the sample point as we expect for a spline interpolation. Experiments with C-MOMS splines shows that in terms of interpolation C-MOMS performs better than B-splines and a little worse than O-MOMS, when O-MOMS is implemented at full precision.

The only thing left to do is to compute the equation for the Farrow resampler. We may use (5.53) to compute the interpolation function $\phi(t) = \beta^3(t) - 1/3d\beta^3(t)/dt$ and then the Farrow structure is sorted according to the delays d^k . Alternatively we can use the precomputed equation (5.60) for the B-splines and apply the differentiation directly to the Farrow matrix, i.e., we compute $c_k^{\text{new}} = c_k - (k+1)c_{k+1}/3$ for $k = 0, 1$, and 2 , since $p(1) = -1/3$ and $p(2) = p(3) = 0$ for cubic C-MOMS. The same principle can also be applied to compute the Farrow equations for O-MOMS and I-MOMS; see Exercise 5.20, p. 414. For C-MOMS this yields the following four equations:

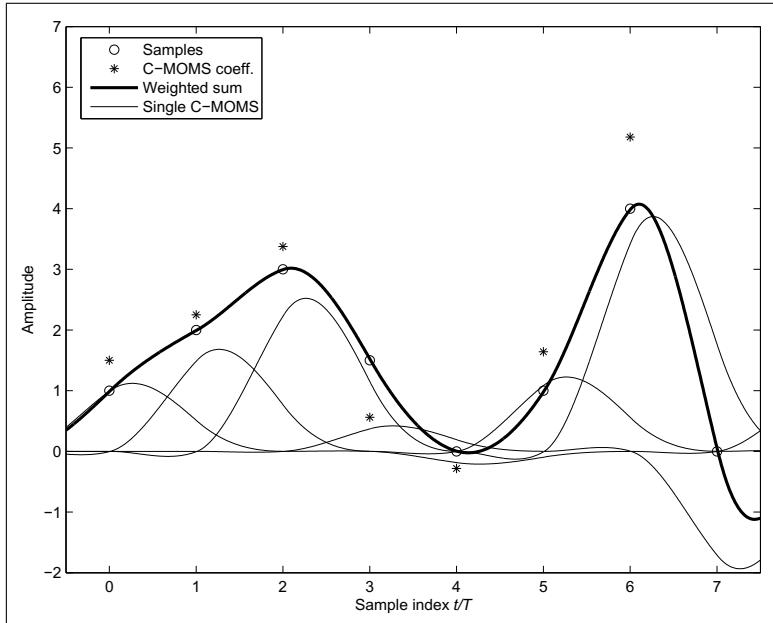


Fig. 5.49. C-MOMS interpolation using IIR compensation filter

$$\begin{aligned}
 d^0 : \quad 0 & \quad +2x(n)/3 + 1x(n-1)/3 = c_0 \\
 d^1 : \quad 0 & \quad +x(n+1)/6 + 2x(n)/3 - 5x(n-1)/6 = c_1 \\
 d^2 : -x(n+2)/6 & \quad +x(n+1) \quad -3x(n)/2 + 2x(n-1)/3 = c_2 \\
 d^3 : x(n+2)/6 & \quad -x(n+1)/2 + x(n)/2 \quad -x(n-1)/6 = c_3
 \end{aligned} \tag{5.64}$$

We now develop the VHDL code for the cubic C-MOMS fractional rate changer.

Example 5.14: $R = 0.75$ Rate Changer V

The following VHDL code¹⁰ shows an $R = 3/4$ rate change using a C-MOMS spline polynomial of degree three:

```

PACKAGE n_bits_int IS
  -- User defined types
  SUBTYPE U2 IS INTEGER RANGE 0 TO 3;
  SUBTYPE U4 IS INTEGER RANGE 0 TO 15;
  SUBTYPE S8 IS INTEGER RANGE -128 TO 127;
  SUBTYPE S9 IS INTEGER RANGE -2**8 TO 2**8-1;
  SUBTYPE S17 IS INTEGER RANGE -2**16 TO 2**16-1;
  TYPE AO_3S8 IS ARRAY (0 TO 3) OF S8;
  TYPE AO_2S9 IS ARRAY (0 TO 2) OF S9;
  TYPE AO_4S17 IS ARRAY (0 TO 4) OF S17;
END n_bits_int;

```

¹⁰ The equivalent Verilog code `cmoms.v` for this example can be found in Appendix A on page 841. Synthesis results are shown in Appendix B on page 881.

```

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-----
ENTITY cmoms IS                               -----> Interface
  GENERIC (IL : INTEGER := 3);-- Input buffer length -1
  PORT (clk      : IN STD_LOGIC; -- System clock
        reset     : IN STD_LOGIC; -- Asynchronous reset
        count_o   : OUT U4; -- Counter FSM
        ena_in_o  : OUT BOOLEAN; -- Sample input enable
        ena_out_o : OUT BOOLEAN; -- Shift output enable
        x_in      : IN S8; -- System input
        xiir_o    : OUT S9; -- IIR filter output
        c0_o, c1_o, c2_o, c3_o : OUT S9; -- C-MOMS matrix
        y_out     : OUT S9); -- System output
END cmoms;
-----
ARCHITECTURE fpga OF cmoms IS

  SIGNAL count  : U4; -- Cycle R_1*R_2
  SIGNAL t       : U2;
  SIGNAL ena_in, ena_out : BOOLEAN; -- FSM enables
  SIGNAL x, ibuf : A0_3S8;           -- TAP registers
  SIGNAL xiir : S9 := 0; -- iir filter output
  -- Precomputed value for d**k :
  CONSTANT d1 : A0_2S9 := (0,85,171);
  CONSTANT d2 : A0_2S9 := (0,28,114);
  CONSTANT d3 : A0_2S9 := (0,9,76);
  -- Spline matrix output:
  SIGNAL c0, c1, c2, c3      : S9;

BEGIN

  FSM: PROCESS (reset, clk)      -----> Control the system
  BEGIN
    IF reset = '1' THEN          -- sample at clk rate
      count <= 0;               -- Asynchronous reset
      t <= 1;
    ELSIF rising_edge(clk) THEN
      IF count = 11 THEN
        count <= 0;
      ELSE
        count <= count + 1;
      END IF;
    CASE count IS
      WHEN 2 | 5 | 8 | 11 =>
        ena_in <= TRUE;
      WHEN others =>
        ena_in <= FALSE;
    END CASE;
  END PROCESS;

```

```

END CASE;
CASE count IS
    WHEN 3 | 7 | 11 =>
        ena_out <= TRUE;
    WHEN others =>
        ena_out <= FALSE;
END CASE;
-- Compute phase delay
IF ENA_OUT THEN
    IF t >= 2 THEN
        t <= 0;
    ELSE
        t <= t + 1;
    END IF;
END IF;
END IF;
END PROCESS FSM;

-- Coeffs: H(z)=1.5/(1+0.5z^-1)
IIR: PROCESS(clk, reset)           -----> Behavioral Style
VARIABLE x1 : S9;
BEGIN -- Compute iir coefficients first
    IF reset = '1' THEN           -- Asynchronous clear
        xiir <= 0; x1 := 0;
    ELSIF rising_edge(clk) THEN   -- iir:
        IF ENA_IN THEN
            xiir <= 3 * x1 / 2 - xiir / 2;
            x1 := x_in;
        END IF;
    END IF;
END PROCESS;

TAP: PROCESS(clk, reset, ENA_IN)
BEGIN
    IF reset = '1' THEN           -- One tapped delay line
        FOR I IN 0 TO IL LOOP
            ibuf(I) <= 0;          -- Clear one
        END LOOP;
    ELSIF rising_edge(clk) THEN
        IF ENA_IN THEN
            FOR I IN 1 TO IL LOOP
                ibuf(I-1) <= ibuf(I);      -- Shift one
            END LOOP;
            ibuf(IL) <= xiir;          -- Input in register IL
        END IF;
    END IF;
END PROCESS;

GET: PROCESS(clk, reset, ENA_OUT)
BEGIN
    IF reset = '1' THEN           -- Get 4 samples at one time
        FOR I IN 0 TO IL LOOP
            x(I) <= 0;           -- Clear one
        END LOOP;
    END IF;
END PROCESS;

```

```

        END LOOP;
ELSIF rising_edge(clk) THEN
    IF ENA_OUT THEN
        FOR I IN 0 TO IL LOOP -- take over input buffer
            x(I) <= ibuf(I);
        END LOOP;
    END IF;
END IF;
END PROCESS;

-- Compute sum-of-products:
SOP: PROCESS (clk, reset, ENA_OUT)
VARIABLE y, y0, y1, y2, y3, h0, h1 : S17;
BEGIN
    -- pipeline registers
    -- Matrix multiplier C-MOMS matrix:
    --   x(0)      x(1)      x(2)      x(3)
    --   0.3333    0.6667    0          0
    --   -0.8333   0.6667   0.1667    0
    --   0.6667   -1.5      1.0       -0.1667
    --   -0.1667   0.5      -0.5      0.1667
    IF reset = '1' THEN -- Asynchronous clear
        c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0;
        y0 := 0; y1 := 0; y2 := 0; y3 := 0;
        y := 0; h0 := 0; h1 := 0;
    ELSIF rising_edge(clk) THEN
        IF ENA_OUT THEN
            c0 <= (85 * x(0) + 171 * x(1))/256;
            c1 <= (171 * x(1) - 213 * x(0) + 43 * x(2)) / 256;
            c2 <= (171 * x(0) - 43 * x(3))/256 - 3*x(1)/2+x(2);
            c3 <= 43 * (x(3) - x(0)) / 256 + (x(1) - x(2))/2;
        -- No Farrow structure, parallel LUT for delays
        -- for u=0:3, y=y+f(u)*d^u; end;
            y := h0 + h1; -- Use pipelined adder tree
            h0 := y0 + y1;
            h1 := y2 + y3;
            y0 := c0 * 256;
            y1 := c1 * d1(t);
            y2 := c2 * d2(t);
            y3 := c3 * d3(t);
        END IF;
    END IF;
    y_out <= y/256; -- Connect to output
END PROCESS SOP;

c0_o <= c0; -- Provide some test signal as outputs
c1_o <= c1;
c2_o <= c2;
c3_o <= c3;
count_o <= count;
ena_in_o <= ena_in;
ena_out_o <= ena_out;
xiir_o <= xiir;

```

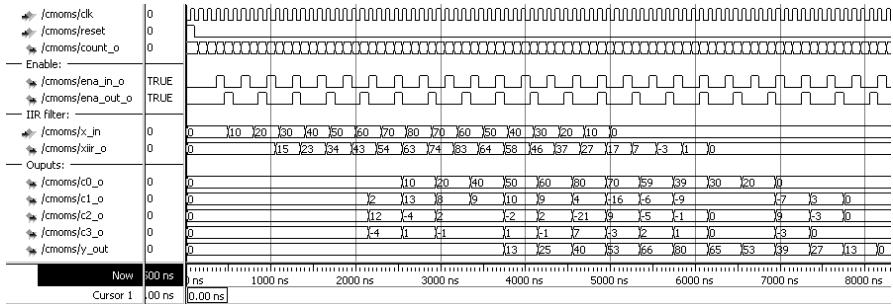


Fig. 5.50. VHDL simulation of the $R = 3/4$ rate change using cubic C-MOMS splines and a one-pole IIR compensation filter

END fpga;

The HDL code for the control is similar to the `rc_sinc` design discussed in Example 5.9, p. 345. The first `PROCESS` is the FSM and includes the control flow and the generation of the enable signals for input and output buffer. The computation of the index for the delay $d_1=d^1$ and its power representation $d_2=d^2$ and $d_3=d^3$ are precomputed and stored in tables as constant. The full round takes 12 clock cycles. The IIR `PROCESS` blocks include the IIR compensation filter. The next two `PROCESS` blocks include the input buffer and the TAP delay line. Note that only one tapped delay line is used for all four polynomial coefficients c_k . The SOP `PROCESS` block includes the cubic C-MOMS matrix computation and the output combiner. Note that no Farrow structure is used to speed up the computation with a parallel multiplier/adder tree structure. This speeds up the design by a factor of 2. The output `y_out` was chosen to have an additional guard bit. The design uses 549 LEs, 3 embedded multipliers, and has an `Fmax=95.27 MHz` registered performance using the `TimeQuest` slow 85C model.

A simulation of the filter is shown in Fig. 5.50. The simulation shows first the control and enable signals of the FSM. A rectangular input `x_in` similar to that in Fig. 5.51 is used. The IIR filter output shows the sharpening of the edges. The C-MOMS matrix output values c_k are weighted by d^k and summed to generate the output `y_out`.

5.14

As with the Lagrange interpolation we may also use a Farrow combiner to compute the output `y_out`. This is particularly interesting if array multipliers are available and we have large values of R_1 and therefore large constant table requirements; see Exercise 5.21 (p. 414).

Finally let us demonstrate the limits of our rate change methods. One particularly difficult problem [153] is the rate change for a rectangular input signal, since we know from the Gibbs phenomenon (see Fig. 3.6, p. 188) that any finite filter has the tendency to introduce ringing at a rectangular edge. Within a DAT recorder two frequencies 32 kHz and 48 kHz are in use and a conversion between them is a common task. The rational rate change factor in this case is $R = 3/2$, if we increase the sampling rate from 32 to 48 kHz. This rate change is shown for a rectangular wave in Fig. 5.51 using O-MOMS

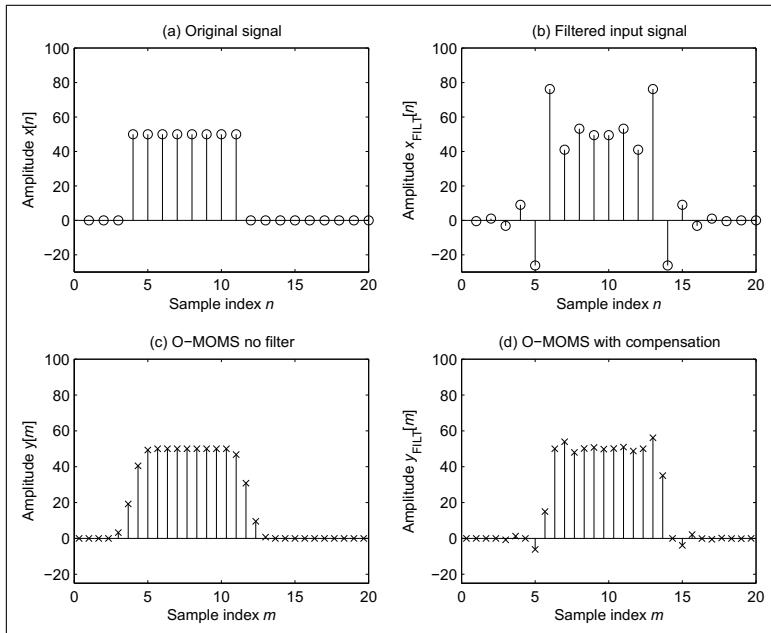


Fig. 5.51. O-MOMS-based fractional $R = 3/2$ rate change. (a) Original signal. (b) Original signal filter with length-11 FIR compensation filter. (c) O-MOMS approximation (no compensation filter). (d) O-MOMS rate change using a compensation filter

spline interpolation. The FIR prefilter output shown in Fig. 5.51b emphasizes the edges. Figure 5.51c shows the result of the O-MOMS cubic spline rate changer without FIR prefILTERing. Although the signal without the filter seems smoother, a closer look reveals that the edges in the O-MOMS cubic spline interpolation are now better preserved than without prefILTERing, as shown in Fig. 5.51d. But it can still be seen that, even with O-MOMS and a length-11 FIR compensation filter at full precision, the Gibbs phenomenon is visible.

5.7 Filter Banks

A *digital filter bank* is a collection of filters having a common input or output, as shown in Fig. 5.52. One common application of the *analysis filter bank* shown in Fig. 5.52a is spectrum analysis, i.e., to split the input signal into R different so-called subband signals. The combination of several signals into a common output signal, as shown in Fig. 5.52b, is called a *synthesis filter bank*. The analysis filter may be nonoverlapping, slightly overlapping, or substantially overlapping. Figure 5.53 shows an example of a slightly overlapping filter bank, which is the most common case.

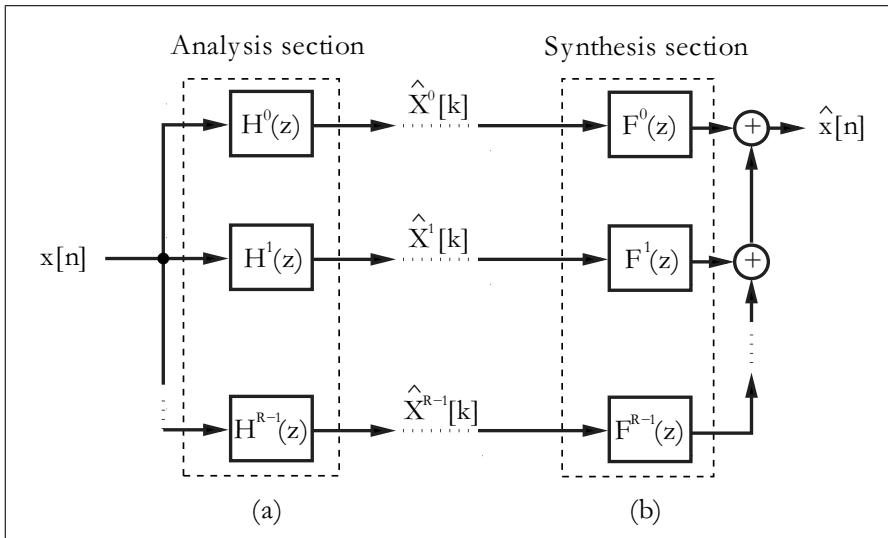


Fig. 5.52. Typical filter bank decomposition system showing (a) analysis, and (b) synthesis filters

Another important characteristic that distinguishes different classes of filter banks is the bandwidth and spacing of the center frequencies of the filters. A popular example of a *nonuniform filter bank* is the octave-spaced or *wavelet filter bank*, which will be discussed in Sect. 5.8 (p. 395). In *uniform filter banks*, all filters have the same bandwidth and sampling rates. From the implementation standpoint, uniform, maximal decimating filter banks are often preferred, because they can be realized with the help of an FFT algorithm, as shown in the next section.

5.7.1 Uniform DFT Filter Bank

In a maximal decimating, or critically sampled filter bank, the decimation or interpolation R is equal to the number of bands K . We call it a *DFT filter bank* if the r^{th} band filter $h^r[n]$ is computed from the “modulation” of a single prototype filter $h[n]$, according to

$$h^r[n] = h[n]W_R^{rn} = h[n]e^{-j2\pi rn/R}. \quad (5.65)$$

An efficient implementation of the R channel filter bank can be generated if we use polyphase decomposition (see Sect. 5.2, p. 309) of the filter $h^r[n]$ and the input signal $x[n]$. Because each of these bandpass filters is critically sampled, we use a decomposition with R polyphase signals according to

$$h[n] = \sum_{k=0}^{R-1} h_k[n] \leftrightarrow h_k[m] = h[mR - k] \quad (5.66)$$

$$x[n] = \sum_{k=0}^{R-1} x_k[n] \leftrightarrow x_k[m] = x[mR - k]. \quad (5.67)$$

If we now substitute (5.66) into (5.65), we find that all bandpass filters $h^r[n]$ share the same polyphase filter $h_k[n]$, while the “twiddle factors” for each filter are different. This structure is shown in Fig. 5.54a for the r^{th} filter $h^r[n]$. It is now obvious that this “twiddle multiplication” for $h^r[n]$ corresponds to the r^{th} DFT component, with an input vector of $\hat{x}_0[n], \hat{x}_1[n], \dots, \hat{x}_{R-1}[n]$. The computation for the whole analysis band can be reduced to filtering with R polyphase filters, followed by a DFT (or FFT) of these R filtered components, as shown in Fig. 5.54b. This is obviously much more efficient than direct computation using the filter defined in (5.65) (see Exercise 5.6, p. 412).

The polyphase filter bank for the uniform DFT *synthesis bank* can be developed as an inverse operation to the analysis bank, i.e., we can use the R spectral components $\hat{X}^r[k]$ as input for the inverse DFT (or FFT), and reconstruct the output signal using a polyphase interpolator structure, shown in Fig. 5.55. The reconstruction bandpass filter becomes

$$f^r[n] = \frac{1}{R} f[n] W_R^{-rn} = f[n] e^{j2\pi rn/R}. \quad (5.68)$$

If we now combine the analysis and synthesis filter banks, we can see that the DFT and IDFT annihilate each other, and *perfect reconstruction* occurs if the convolution of the included polyphase filter gives a unit sample function, i.e.,

$$h_r[n] * f_r[n] = \begin{cases} 1 & n = d \\ 0 & \text{else.} \end{cases} \quad (5.69)$$

In other words, the two polyphase functions must be inverse filters of each other, i.e.,

$$H_r(z) \times F_r(z) = z^{-d}$$

$$F_r(z) = \frac{z^{-d}}{H_r(z)},$$

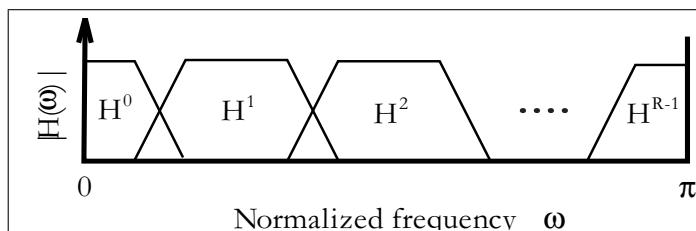


Fig. 5.53. R channel filter bank, with a small amount of overlapping

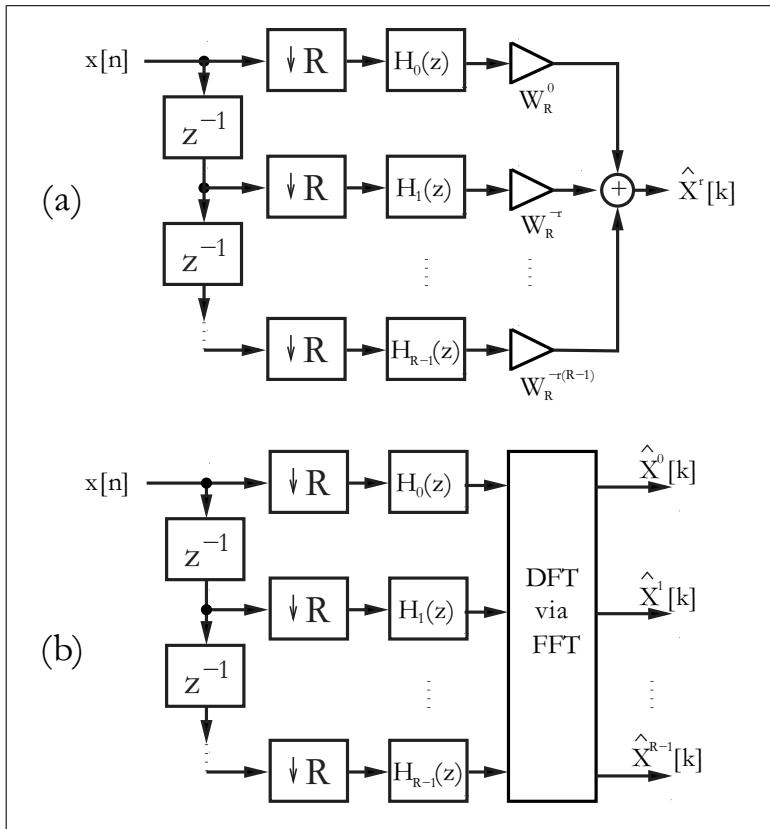


Fig. 5.54. (a) Analysis DFT filter bank for channel k . (b) Complete analysis DFT filter bank

where we allow a delay d in order to have causal (realizable) filters. In a practical design, these ideal conditions cannot be met exactly by two FIR filters. We can use approximation for the two FIR filters, or we can combine an FIR and IIR, as shown in the following example.

Example 5.15: DFT Filter Bank

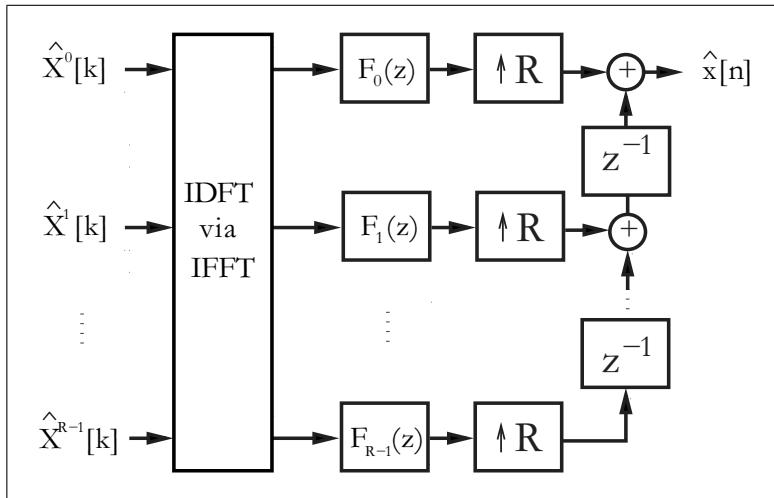
The *lossy integrator* studied in Example 4.3 (p. 241) should be interpreted in the context of a DFT filter bank with $R = 2$. The difference equation was

$$y[n+1] = \frac{3}{4}y[n] + x[n]. \quad (5.70)$$

The impulse response of this filter in the z -domain is

$$F(z) = \frac{z^{-1}}{1 - 0.75z^{-1}}. \quad (5.71)$$

In order to get two polyphase filters, we use a similar scheme as for the “scattered look-ahead” modification (see Example 4.5, p. 244), i.e., we introduce

**Fig. 5.55.** DFT synthesis filter bank

an additional pole/zero pair at the mirror position. Multiplying numerator and denominator by $(1 + 0.75z^{-1})$ yields

$$F(z) = \underbrace{\frac{0.75z^{-2}}{1 - 0.75^2z^{-2}}}_{H_0(z^2)} + z^{-1} \underbrace{\frac{1}{1 - 0.75^2z^{-2}}}_{H_1(z^2)} \quad (5.72)$$

$$= H_0(z^2) + z^{-1} H_1(z^2), \quad (5.73)$$

which gives the two polyphase filters:

$$H_0(z) = \frac{0.75z^{-1}}{1 - 0.75^2z^{-1}} = 0.75z^{-1} + 0.4219z^{-2} + 0.2373z^{-3} + \dots \quad (5.74)$$

$$H_1(z) = \frac{1}{1 - 0.75^2z^{-1}} = 1 + 0.5625z^{-1} + 0.3164z^{-2} + \dots. \quad (5.75)$$

We can approximate these impulse responses with a nonrecursive FIR, but to get less than 1% error we must use about 16 coefficients. It is therefore much more efficient if we use the two recursive polyphase IIR filters defined by (5.74) and (5.75). After decomposition with the polyphase filters, we then apply a 2-point DFT, which is given by

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

The whole analysis filter bank can now be constructed as shown in 5.56a. For the synthesis bank, we first compute the inverse DFT using

$$\mathbf{W}^{-1} = \frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

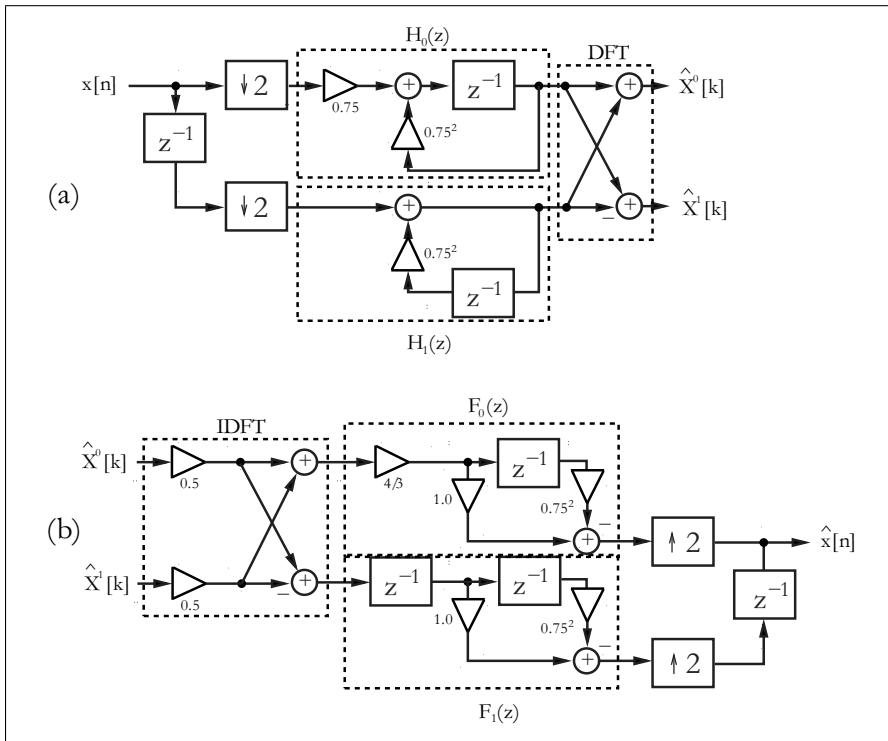


Fig. 5.56. Critically sampled uniform DFT filter bank for $R = 2$. (a) Analysis filter bank. (b) Synthesis filter bank

In order to get a perfect reconstruction we must find the inverse polyphase filter to $h_0[n]$ and $h_1[n]$. This is not difficult, because the $H_r(z)$'s are single-pole IIR filters, and $F_r(z) = z^{-d}/H_r(z)$ must therefore be two-tap FIR filters. Using (5.74) and (5.75), we find that $d = 1$ is already sufficient to get causal filters, and it is

$$F_0[n] = \frac{4}{3} (1 - 0.75^2 z^{-1}) \quad (5.76)$$

$$F_1[n] = z^{-1} - 0.75^2 z^{-2}. \quad (5.77)$$

The synthesis bank is graphically interpreted in Fig. 5.56b. 5.15

5.7.2 Two-channel Filter Banks

Two-channel filter banks are an important tool for the design of general filter banks and wavelets. Figure 5.57 shows an example of a two-channel filter bank that splits the input $x[n]$ using lowpass ($G(z)$) and highpass ($H(z)$)

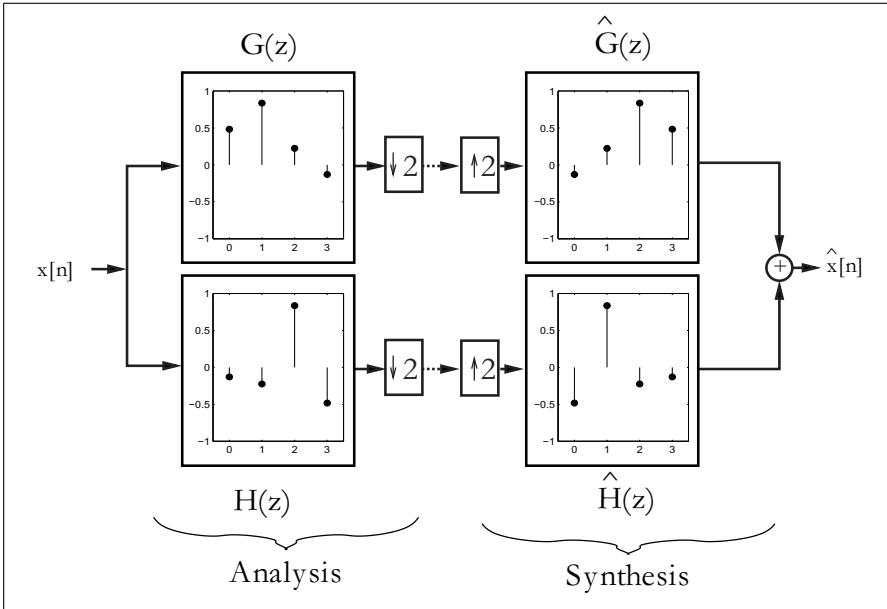


Fig. 5.57. Two-channel filter bank using Daubechies filter of length-4

“analysis” filters. The resulting signal $\hat{x}[n]$ is reconstructed using lowpass and highpass “synthesis” filters. Between the analysis and synthesis sections are decimation and interpolation by 2 units. The signal between the decimators and interpolators is often quantized, and nonlinearly processed for enhancement, or compressed.

It is common practice to define only the lowpass filter $G(z)$, and to use its definition to specify the highpass filter $H(z)$. The construction rule is normally given by

$$h[n] = (-1)^n g[n] \circledast H(z) = G(-z), \quad (5.78)$$

which defines the filters to be mirrored pairs. Specifically, in the frequency domain, $|H(e^{j\omega})| = |G(e^{j(\omega-\pi)})|$. This is a *quadrature mirror filter (QMF)* bank, because the two filters have mirror symmetry to $\pi/2$.

For the synthesis shown in Fig. 5.57, we first use an expander (a sampling rate increase of 2), and then two separate reconstruction filters, $\hat{G}(z)$ and $\hat{H}(z)$, to reconstruct $\hat{x}[n]$. A challenging question now is, can the input signal be perfectly reconstructed, i.e., can we satisfy

$$\hat{x}[n] = x[n - d]? \quad (5.79)$$

That is, a perfectly reconstructed signal has the same shape as the original, up to a phase (time) shift. Because $G(z)$ and $H(z)$ are not ideal rectangular filters, achieving perfect reconstruction is not a trivial problem. Both filters produce essential aliasing components after the downsampling by 2, as

shown in Fig. 5.57. The simple orthogonal filter bank that satisfies (5.79) is attributed to Alfred Haar (circa 1910) [154].

Example 5.16: Two-Channel Haar Filter Bank I

The filter transfer functions of the two-channel QMF filter bank from Fig. 5.58 are¹²

$$\begin{aligned} G(z) &= 1 + z^{-1} & H(z) &= 1 - z^{-1} \\ \hat{G}(z) &= \frac{1}{2}(1 + z^{-1}) & \hat{H}(z) &= \frac{1}{2}(-1 + z^{-1}). \end{aligned}$$

Using data found in the table in Fig. 5.58, it can be verified that the filter produces a perfect reconstruction of the input. The input sequence $x[0], x[1], x[2], \dots$, processed by $G(z)$ and $H(z)$, yields the sum $x[n] + x[n-1]$ and difference $x[n] - x[n-1]$, respectively. The downsampling followed by upsampling forces every second value to zero. After applying the synthesis filter and combining the output we again get the input sequence delayed by one, i.e., $\hat{x}[n] = x[n-1]$, a *perfect* reconstruction with $d = 1$. 5.16

In the following we will discuss the general relationships the four filters must obey to get a perfect reconstruction. It is useful to remember that decimation and interpolation by 2 of a signal $s[k]$ is equivalent to multiplying $S(z)$ by the sequence $\{1, 0, 1, 0, \dots\}$. This translates, in the z -domain, to

$$S_{\downarrow\uparrow}(z) = \frac{1}{2} (S(z) + S(-z)). \quad (5.80)$$

If this signal is applied to the two-channel filter bank, the lowpass path $X_{\downarrow\uparrow G}(z)$ and highpass path $X_{\downarrow\uparrow H}(z)$ become

$$X_{\downarrow\uparrow G}(z) = \frac{1}{2} (X(z)G(z) + X(-z)G(-z)), \quad (5.81)$$

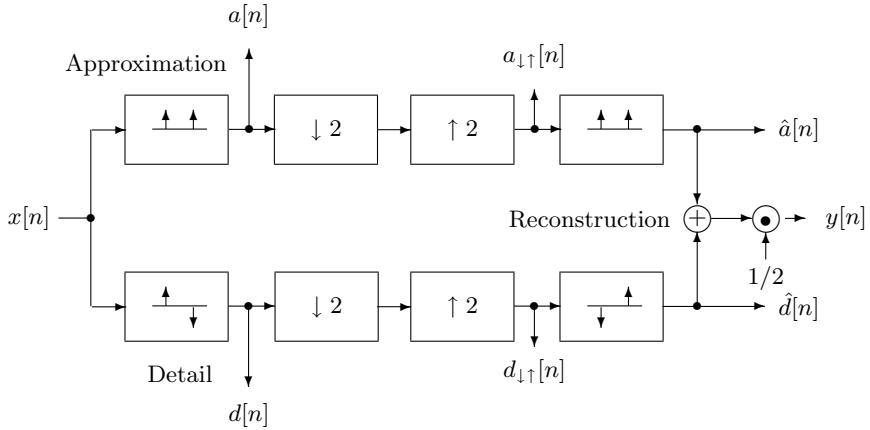
$$X_{\downarrow\uparrow H}(z) = \frac{1}{2} (X(z)H(z) + X(-z)H(-z)). \quad (5.82)$$

After multiplication by the synthesis filter $\hat{G}(z)$ and $\hat{H}(z)$, and summation of the results, we get $\hat{X}(z)$ as

$$\begin{aligned} \hat{X}(z) &= X_{\downarrow\uparrow G}(z)\hat{G}(z) + X_{\downarrow\uparrow H}(z)\hat{H}(z) \\ &= \frac{1}{2} \left(G(z)\hat{G}(z) + H(z)\hat{H}(z) \right) X(z) \\ &\quad + \frac{1}{2} \left(G(-z)\hat{G}(z) + H(-z)\hat{H}(z) \right) X(-z). \end{aligned} \quad (5.83)$$

The factor of $X(-z)$ shows the aliasing component, while the term at $X(z)$ shows the amplitude distortion. For a *perfect* reconstruction this translates into the following:

¹² Sometimes the amplitude factors are chosen in such a way that *orthonormal* filters are obtained, i.e., $\sum_n |h[n]|^2 = 1$. In this case, the filters have an amplitude factor of $1/\sqrt{2}$. This will complicate a hardware design significantly.



	Time step				
	0	1	2	3	4
$x[n]$	$x[0]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$
$a[n]$	$x[0]$	$x[0] + x[1]$	$x[1] + x[2]$	$x[2] + x[3]$	$x[3] + x[4]$
$d[n]$	$x[0]$	$x[1] - x[0]$	$x[2] - x[1]$	$x[3] - x[2]$	$x[4] - x[3]$
$a_{\downarrow\uparrow}[n]$	$x[0]$	0	$x[1] + x[2]$	0	$x[3] + x[4]$
$d_{\downarrow\uparrow}[n]$	$x[0]$	0	$x[2] - x[1]$	0	$x[4] - x[3]$
$\hat{a}[n]$	$x[0]$	$x[0]$	$x[1] + x[2]$	$x[1] + x[2]$	$x[3] + x[4]$
$\hat{d}[n]$	$-x[0]$	$x[0]$	$x[1] - x[2]$	$x[2] - x[1]$	$x[3] - x[4]$
$\hat{x}[n]$	0	$x[0]$	$x[1]$	$x[2]$	$x[3]$

Fig. 5.58. Two-channel Haar-QMF bank

Theorem 5.17: Perfect Reconstruction

A perfect reconstruction for a two-channel filter bank, as shown in Fig. 5.57, is achieved if

- 1) $G(-z)\hat{G}(z) + H(-z)\hat{H}(z) = 0$, i.e., the reconstruction is free of aliasing.
- 2) $G(z)\hat{G}(z) + H(z)\hat{H}(z) = 2z^{-d}$, i.e., the amplitude distortion has amplitude one.

Let us check this condition for the Haar filter bank.

Example 5.18: Two-Channel Haar Filter Bank II

The filters of the two-channel Haar QMF bank were defined by

$$\begin{aligned} G(z) &= 1 + z^{-1} & H(z) &= 1 - z^{-1} \\ \hat{G}(z) &= \frac{1}{2}(1 + z^{-1}) & \hat{H}(z) &= \frac{1}{2}(-1 + z^{-1}). \end{aligned}$$

The two conditions from Theorem 5.17 can be proved with:

$$\begin{aligned}
1) \quad & G(-z)\hat{G}(z) + H(-z)\hat{H}(z) \\
&= \frac{1}{2}(1 - z^{-1})(1 + z^{-1}) + \frac{1}{2}(1 + z^{-1})(-1 + z^{-1}) \\
&= \frac{1}{2}(1 - z^{-2}) + \frac{1}{2}(-1 + z^{-2}) = 0 \quad \checkmark \\
2) \quad & G(z)\hat{G}(z) + H(z)\hat{H}(z) \\
&= \frac{1}{2}(1 + z^{-1})^2 + \frac{1}{2}(1 - z^{-1})(-1 + z^{-1}) \\
&= \frac{1}{2}((1 + 2z^{-1} + z^{-2}) + (-1 + 2z^{-1} - z^{-2})) = 2z^{-1} \quad \checkmark
\end{aligned}$$

5.18

For the proof using Theorem 5.17, it can be noted that the perfect reconstruction condition does not change if we switch the analysis and synthesis filters.

In the following we will discuss some restrictions that can be made in the filter design to fulfill the condition from Theorem 5.17 more easily.

First, we limit the filter choice by using the following:

Theorem 5.19: Aliasing-Free Two-Channel Filter Bank

A two-channel filter bank is *aliasing-free* if

$$G(-z) = -\hat{H}(z) \quad \text{and} \quad H(-z) = \hat{G}(z). \quad (5.84)$$

This can be checked if we use (5.84) for the first condition of Theorem 5.17. Using a length-4 filter, these two conditions can be interpreted as follows:

$$\begin{aligned}
g[n] &= \{g[0], g[1], g[2], g[3]\} \rightarrow \hat{h}[n] = \{-g[0], g[1], -g[2], g[3]\} \\
h[n] &= \{h[0], h[1], h[2], h[3]\} \rightarrow \hat{g}[n] = \{h[0], -h[1], h[2], -h[3]\}.
\end{aligned}$$

With the restriction of the filters as in Theorem 5.19, we can now simplify the second condition in Theorem 5.17. It is useful to define first an auxiliary *product filter* $F(z) = G(z)\hat{G}(z)$. The second condition from Theorem 5.17 becomes

$$G(z)\hat{G}(z) + H(z)\hat{H}(z) = F(z) - \hat{G}(-z)G(-z) = F(z) - F(-z) \quad (5.85)$$

and we finally get

$$F(z) - F(-z) = 2z^{-d}, \quad (5.86)$$

i.e., the product filter must be a *half-band filter*.¹³ The construction of a perfect reconstruction filter bank uses the following three simple steps:

¹³ For the definition of a half-band filter, see p. 339.

Algorithm 5.20: Perfect-Reconstruction Two-Channel Filter Bank

- 1) Define a normalized causal half-band filter according to (5.86).
- 2) Factor the filter $F(z)$ in $F(z) = G(z)\hat{G}(z)$.
- 3) Compute $H(z)$ and $\hat{H}(z)$ using (5.84), i.e., $\hat{H}(z) = -G(-z)$ and $H(z) = \hat{G}(-z)$.

We wish to demonstrate Algorithm 5.20 with the following example. To simplify the notation we will, in the following example, write a combination of a length L filter for $G(z)$, and length N for $\hat{G}(z)$, as an L/N filter.

Example 5.21: Perfect-Reconstruction Filter Bank Using F3

The (normalized) causal half-band filter F3 (Table 5.3, p. 339) of length seven has the following z -domain transfer function:

$$F3(z) = \frac{1}{16} (-1 + 9z^{-2} + 16z^{-3} + 9z^{-4} - z^{-6}). \quad (5.87)$$

Using (5.86) we first verify that $F3(z) - F3(-z) = 2z^{-3}$. The zeros of the transfer function are at $z_{01-4} = -1$, $z_{05} = 2 + \sqrt{3} = 3.7321$, and $z_{06} = 2 - \sqrt{3} = 0.2679 = 1/z_{05}$. There are different choices for factoring $F(z) = G(z)\hat{G}(z)$. A 5/3 filter is, for instance,

- a) $G(z) = (-1 + 2z^{-1} + 6z^{-2} + 2z^{-3} - z^{-4})/8$ and $\hat{G}(z) = (1 + 2z^{-1} + z^{-2})/2$. We may design a 4/4 filter as:

b) $G(z) = \frac{1}{4}(1 + z^{-1})^3$ and $\hat{G}(z) = \frac{1}{4}(-1 + 3z^{-1} + 3z^{-2} - z^{-3})$.

Another configuration of the 4/4 configuration uses the Daubechies filter configuration, which is often found in wavelet applications and has the form:

c) $G(z) = \frac{1-\sqrt{3}}{4\sqrt{2}}(1 + z^{-1})^2(-z_{05} + z^{-1})$ and $\hat{G}(z) = -\frac{1+\sqrt{3}}{4\sqrt{2}}(1 + z^{-1})^2(-z_{06} + z^{-1})$.

Figure 5.59 shows these three combinations, along with their pole/zero plots.

5.21

For the Daubechies filter, the condition $H(z) = -z^{-N}G(-z^{-1})$ holds in addition, i.e., highpass and lowpass polynomials are mirror versions of each other. This is a typical behavior in *orthogonal* filter banks.

From the pole/zero plots shown in Fig. 5.59, for $F(z) = G(z)\hat{G}(z)$ the following conclusions can be made:

Corollary 5.22: Factorization of a Half-Band Filter

- 1) To construct a *real* filter, we must always group the conjugate symmetric zeros at $(z_0$ and $z_0^*)$ in the same filter.
- 2) For *linear-phase* filters, the pole/zero plot must be symmetrical to the unit circle ($z = 1$). Zero pairs at $(z_0$ and $1/z_0$) must be assigned to the *same* filter.
- 3) To have *orthogonal* filters that are mirror polynomials of each other, ($F(z) = U(z)U(z^{-1})$), all pairs z_0 and $1/z_0$ must be assigned to *different* filters.

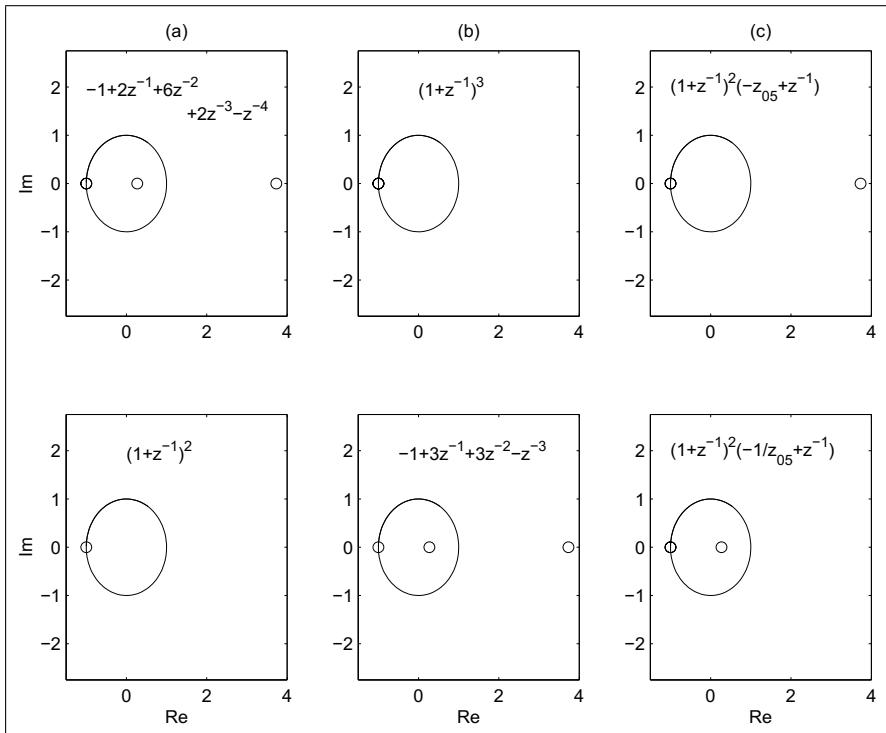


Fig. 5.59. Pole/zero plot for different factorization of the half-band filter F3. Upper row $G(z)$, lower row $\hat{G}(z)$. (a) Linear-phase 5/3 filter. (b) Linear-phase 4/4 filter. (c) 4/4 Daubechies filter

We note that some of the above conditions can *not* be fulfilled at the same time. In particular, rules 2 and 3 represent a contradiction. Orthogonal, linear-phase filters are, in general, not possible, except when all zeros are on the unit circle, as in the case of the Haar filter bank.

If we classify the filter banks from Example 5.21, we find that configurations (a) and (b) are real linear-phase filters, while (c) is a real orthogonal filter.

Implementing Two-Channel Filter Banks

We will now discuss different options for implementing two-channel filter banks. We will first discuss the general case, and then special simplifications that are possible if the filters are QMF, linear-phase, or orthogonal. We will only discuss the analysis filter bank, as synthesis may be achieved with graph transposition.

Polyphase two-channel filter banks. In the general case, with two filters $G(z)$ and $H(z)$, we can realize each filter as a polyphase filter

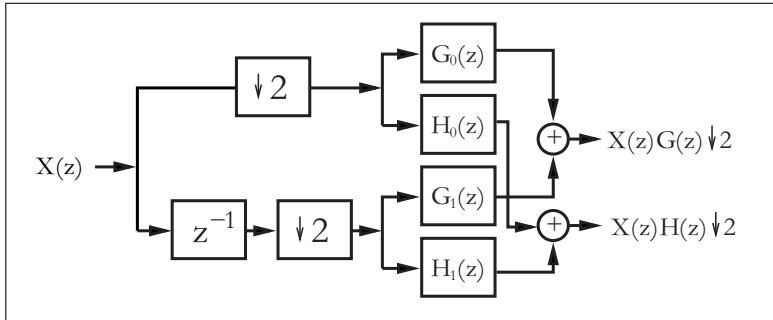


Fig. 5.60. Polyphase implementation of the two-channel filter bank

$$H(z) = H_0(z^2) + z^{-1}H_1(z^2) \quad G(z) = G_0(z^2) + z^{-1}G_1(z^2), \quad (5.88)$$

which is shown in Fig. 5.60. This does not reduce the hardware effort ($2L$ multipliers and $2(L - 1)$ adders are still used), but the design can be run with twice the usual sampling frequency, $2f_s$.

These four polyphase filters have only half the length of the original filters. We may implement these length $L/2$ filters directly or with one of the following methods:

- 1) Run-length filter using short Winograd convolution algorithms [125], discussed in Sect. 5.2.2, p. 315.
- 2) Fast convolution using FFT (discussed in Chap. 6).
- 3) Using advanced arithmetic concepts discussed in Chap. 3, such as distribute arithmetic, reduced adder graph, or residue number system.

Using the fast convolution FFT techniques has the additional benefit that the forward transform for each polyphase filter need only be done once, and also, the inverse transform can be applied to the spectral sum of the two components, as shown in Fig. 5.61. But, in general, FFT methods only give improvements for longer filters, typically, larger than 32; however, the typical two-channel filter length is less than 32.

Lifting. Another general approach to constructing fast and efficient two-channel filter banks is the *lifting* scheme introduced by Swelden [155] and Herley and Vetterli [156]. The basic idea is the use of cross-terms (called lifting and dual-lifting), as in a lattice filter, to construct a longer filter from a short filter, while preserving the perfect reconstruction conditions. The basic structure is shown in Fig. 5.62.

Designing a lifting scheme typically starts with the “lazy filter bank,” with $G(z) = \hat{H}(z) = 1$ and $H(z) = \hat{G}(z) = z^{-1}$. This channel bank fulfills both conditions from Theorem 5.17 (p. 382), i.e., it is a perfect reconstruction filter bank. The following question arises: if we keep one filter fixed, what are filters $S(z)$ and $T(z)$ such that the filter bank is still a perfect reconstruction? The answer is important, and not trivial:

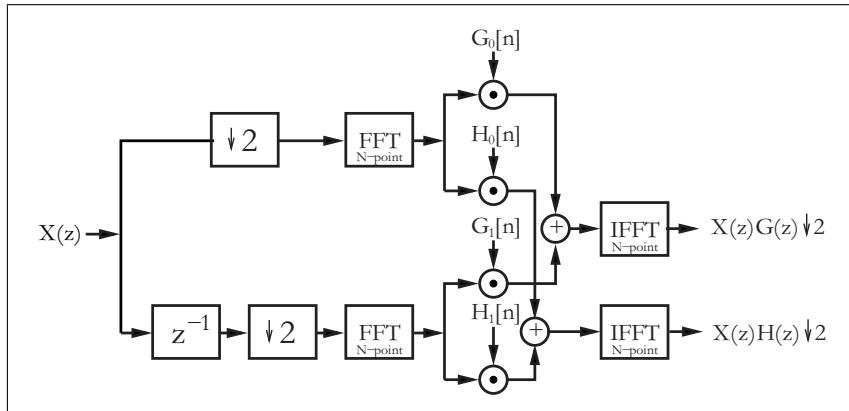


Fig. 5.61. Two-channel filter bank with polyphase decomposition and fast convolution using the FFT. (© Springer Press [5])

$$\text{Lifting: } G'(z) = G(z) + \hat{G}(-z)S(z^2) \quad \text{for any } S(z^2). \quad (5.89)$$

$$\text{Dual-Lifting: } \hat{G}'(z) = \hat{G}(z) + G(-z)T(z^2) \quad \text{for any } T(z^2). \quad (5.90)$$

To check, if we substitute the lifting equation into the perfect reconstruction condition from Theorem 5.17 (p. 382), and we see that both conditions are fulfilled if $\hat{G}(z)$ and $\hat{H}(z)$ still meet the conditions of Theorem 5.19 (p. 383) for the aliasing free filter bank (Exercise 5.9, p. 412).

The conversion of the Daubechies length-4 filter bank into lifting steps demonstrates the design.

Example 5.23: Lifting Implementation of the DB4 Filter

One filter configuration in Example 5.21 (p. 384) was the Daubechies length-4 filter [157, p. 195]. The filter coefficients were

$$G(z) = ((1 + \sqrt{3}) + (3 + \sqrt{3})z^{-1} + (3 - \sqrt{3})z^{-2} + (1 - \sqrt{3})z^{-3}) \frac{1}{4\sqrt{2}}$$

$$H(z) =$$

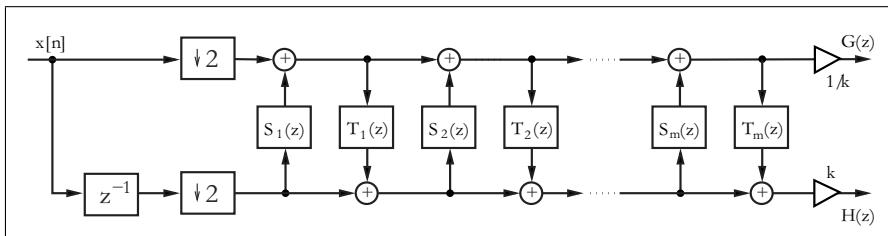


Fig. 5.62. Two-channel filter implementation using lifting and dual-lifting steps

$$(-(1 - \sqrt{3}) + (3 - \sqrt{3})z^{-1} - (3 + \sqrt{3})z^{-2} + (1 + \sqrt{3})z^{-3}) \frac{1}{4\sqrt{2}}.$$

A possible implementation uses two lifting steps and one dual-lifting step. The differential equations that produce a two-channel filter bank based on the above equation are

$$\begin{aligned} h_1[n] &= x[2n+1] - \sqrt{3}x[2n] \\ g_1[n] &= x[2n] + \frac{\sqrt{3}}{4}h_1[n] + \frac{\sqrt{3}-2}{4}h_1[n-1] \\ h_2[n] &= h_1[n] + g_1[n+1] \\ g[n] &= \frac{\sqrt{3}+1}{\sqrt{2}}g_1[n] \\ h[n] &= \frac{\sqrt{3}-1}{\sqrt{2}}h_2[n]. \end{aligned}$$

Note that the early decimation and splitting of the input into even $x[2n]$ and odd $x[2n-1]$ sequences allows the filter to run with $2f_s$. This structure can be directly translated into hardware and can be implemented using Quartus II (Exercise 5.10, p. 412). The implementation will use five multiplications and four adders. The reconstruction filter bank can be constructed based on graph transposition, which is, in the case of the differential equations, a reversing of the operations and flipping of the signs.

5.23

Daubechies and Sweldens [158], have shown that *any* (bi)orthogonal wavelet filter bank can be converted into a sequence of lifting and dual-lifting steps. The number of multipliers and adders required then depends on the number of lifting steps (more steps gives less complexity) and can reach up to 50% compared with the direct polyphase implementation. This approach seems especially promising if the bit width of the multiplier is small [159]. On the other hand, the lattice-like structure does not allow use of reduced adder graph (RAG) techniques, and for longer filters the direct polyphase approach will often be more efficient.

Although the techniques (polyphase decomposition and lifting) discussed so far improve speed or size and cover all types of two-channel filters, additional savings can be achieved if the filters are QMF, linear-phase, or orthogonal. This will be discussed in the following.

QMF implementation. For QMF [160] we have found that according to (5.78),

$$h[n] = (-1)^n g[n] \circledcirc H(z) = G(-z). \quad (5.91)$$

But this implies that the polyphase filters are the same (except the sign), i.e.,

$$G_0(z) = H_0(z) \quad G_1(z) = -H_1(z). \quad (5.92)$$

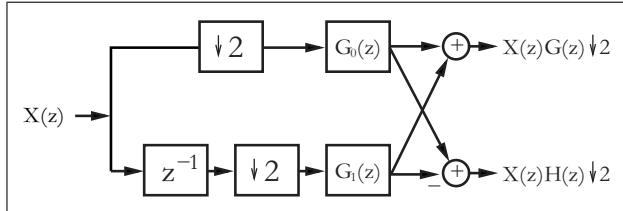


Fig. 5.63. Polyphase realization of the two-channel QMF bank. (© Springer Press [5])

Instead of the four filters from Fig. 5.60, for QMF we only need two filters and an additional “Butterfly,” as shown in Fig. 5.63. This saves about 50%. For the QMF filter we need:

$$L \text{ real adders} \quad L \text{ real multipliers}, \quad (5.93)$$

and the filter can run with twice the usual input-sampling rate.

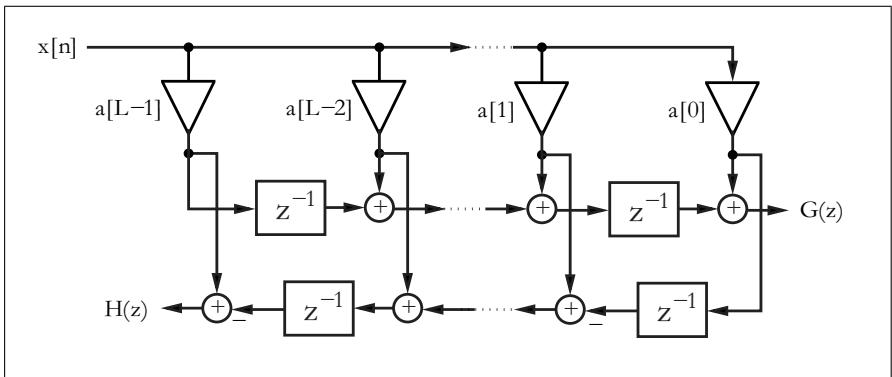


Fig. 5.64. Orthogonal two-channel filter bank using the transposed FIR structure

Orthogonal filter banks. An orthogonal filter pair¹⁴ obeys the conjugate mirror filter (CQF) [161] condition, defined by

$$H(z) = z^{-N} G(-z^{-1}). \quad (5.94)$$

If we use the transposed FIR filter shown in Fig. 5.64, we need only half the number of multipliers. The disadvantage is that we can *not* benefit from polyphase decomposition to double the speed.

Another alternative is realization of the CQF bank using the lattice filter shown in Fig. 5.65. The following example demonstrates the conversion of the direct FIR filter into a lattice filter.

¹⁴ The orthogonal filter name comes from the fact that the scalar product of the filters, for a shift by two (i.e., $\sum g[k]h[k - 2l] = 0, k, l \in \mathbb{Z}$), is zero.

Example 5.24: Lattice Daubechies $L = 4$ Filter Implementation

One filter configuration in Example 5.21 (p. 384) was the Daubechies length-4 filter [157, p. 195]. The filter coefficients were

$$\begin{aligned} G(z) &= \frac{(1 + \sqrt{3}) + (3 + \sqrt{3})z^{-1} + (3 - \sqrt{3})z^{-2} + (1 - \sqrt{3})z^{-3}}{4\sqrt{2}} \\ &= 0.48301 + 0.8365z^{-1} + 0.2241z^{-2} - 0.1294z^{-3} \end{aligned} \quad (5.95)$$

$$\begin{aligned} H(z) &= \frac{-(1 - \sqrt{3}) + (3 - \sqrt{3})z^{-1} - (3 + \sqrt{3})z^{-2} + (1 + \sqrt{3})z^{-3}}{4\sqrt{2}} \\ &= 0.1294 + 0.2241z^{-1} - 0.8365z^{-2} + 0.48301z^{-3}. \end{aligned} \quad (5.96)$$

The transfer function for a two-channel lattice with two stages is

$$G(z) = (1 + a[0]z^{-1} - a[0]a[1]z^{-2} + a[1]z^{-3}) s \quad (5.97)$$

$$H(z) = (-a[1] - a[0]a[1]z^{-1} - a[0]z^{-2} + z^{-3}) s. \quad (5.98)$$

If we now compare (5.95) with (5.97) we find

$$s = \frac{1 + \sqrt{3}}{4\sqrt{2}} \quad a[0] = \frac{3 + \sqrt{3}}{4\sqrt{2}s} \quad a[1] = \frac{1 - \sqrt{3}}{4\sqrt{2}s}. \quad (5.99)$$

We can now translate this structure direct into hardware and implement the filter bank with Quartus II as shown in the following VHDL¹⁵ code:

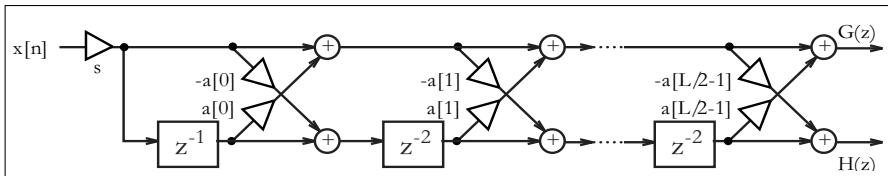


Fig. 5.65. Lattice realization for the orthogonal two-channel filter bank. (© Springer Press [5])

```

PACKAGE n_bits_int IS
    -- User defined types
    SUBTYPE S8 IS INTEGER RANGE -128 TO 127;
    SUBTYPE S9 IS INTEGER RANGE -2**8 TO 2**8-1;
    SUBTYPE S17 IS INTEGER RANGE -2**16 TO 2**16-1;
    TYPE A0_3S17 IS ARRAY (0 TO 3) OF S17;
END n_bits_int;

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

```

¹⁵ The equivalent Verilog code db4latti.v for this example can be found in Appendix A on page 845. Synthesis results are shown in Appendix B on page 881.

```

USE ieee.std_logic_unsigned.ALL;
-----
ENTITY db4latti IS          -----> Interface
  PORT (clk      : IN STD_LOGIC;   -- System clock
        reset    : IN STD_LOGIC;   -- Asynchronous reset
        clk2     : OUT STD_LOGIC;  -- Clock divider
        x_in     : IN S8;         -- System input
        x_e, x_o : OUT S17;      -- Even/odd x input
        g, h     : OUT S9);      -- g/h filter output
END db4latti;
-----
ARCHITECTURE fpga OF db4latti IS

  TYPE STATE_TYPE IS (even, odd);
  SIGNAL state           : STATE_TYPE;
  SIGNAL clk_div2        : STD_LOGIC;
  SIGNAL sx_up, sx_low, x_wait : S17 := 0;
  SIGNAL sxa0_up, sxa0_low : S17 := 0;
  SIGNAL up0, up1, low0, low1 : S17 := 0;

BEGIN

  Multiplex: PROCESS (reset, clk) ----> Split into even and
  BEGIN                                     -- odd samples at clk rate
    IF reset = '1' THEN                   -- Asynchronous reset
      state <= even;
      sx_up <= 0; sx_low <= 0;
      clk_div2 <= '0'; x_wait <= 0;
    ELSIF rising_edge(clk) THEN
      CASE state IS
        WHEN even =>
          -- Multiply with 256*s=124
          sx_up  <= 4 * (32 * x_in - x_in);
          sx_low <= 4 * (32 * x_wait - x_wait);
          clk_div2 <= '1';
          state <= odd;
        WHEN odd =>
          x_wait <= x_in;
          clk_div2 <= '0';
          state <= even;
      END CASE;
    END IF;
  END PROCESS;

----- Multipliy a[0] = 1.7321
  sxa0_up  <= (2*sx_up  - sx_up /4)
              - (sx_up /64 + sx_up/256);
  sxa0_low <= (2*sx_low - sx_low/4)
              - (sx_low/64 + sx_low/256);
----- First stage -- FF in lower tree
  up0  <= sxa0_low + sx_up;
LowerTreeFF: PROCESS(reset, clk, clk_div2)
BEGIN

```

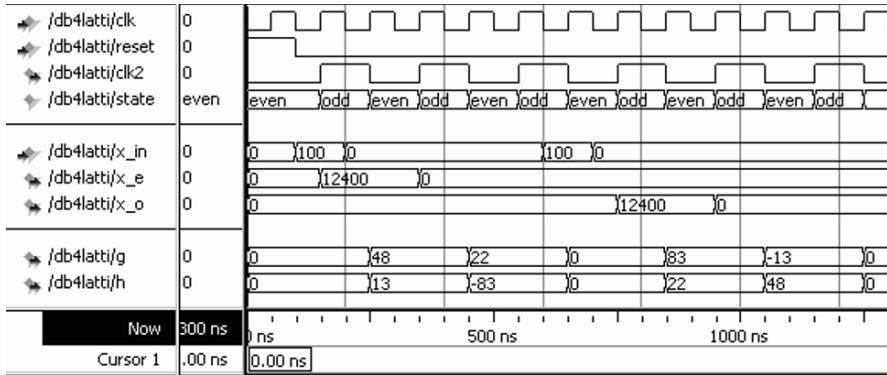


Fig. 5.66. VHDL simulation of the Daubechies length-4 lattice filter bank

```

        IF reset = '1' THEN          -- Asynchronous clear
            low0 <= 0;
        ELSIF rising_edge(clk) THEN
            IF clk_div2 = '1' THEN
                low0 <= sx_low - sxa0_up;
            END IF;
        END IF;
    END PROCESS;

----- Second stage  a[1]=0.2679
up1  <= (up0 - low0/4) - (low0/64 + low0/256);
low1 <= (low0 + up0/4) + (up0/64 + up0/256);

x_e  <= sx_up;   -- Provide some extra test signals
x_o  <= sx_low;
clk2 <= clk_div2;

OutputScale: PROCESS(reset, clk, clk_div2)
BEGIN
    IF reset = '1' THEN          -- Asynchronous clear
        g <= 0; h <= 0;
    ELSIF rising_edge(clk) THEN
        IF clk_div2 = '1' THEN
            g <= up1 / 256;
            h <= low1 / 256;
        END IF;
    END IF;
END PROCESS;

END fpga;

```

This VHDL code is a direct translation of the lattice shown in Fig. 5.65. The incoming stream is multiplied by $s = 0.48 \approx 124/256$. Next, the cross-term product multiplications, with $a[0] = 1.73 \approx (2 - 2^{-2} - 2^{-6} - 2^{-8})$, of the first stage are computed. It follows that the stage 1 additions and the lower tree signal must be delayed by one sample. In the second stage, the cross multiplication by $a[1] = 0.27 \approx (2^{-2} + 2^{-6} + 2^{-8})$ and the final output

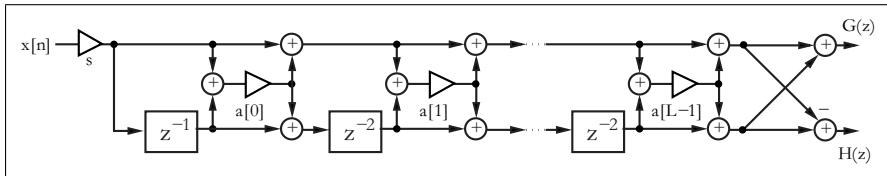


Fig. 5.67. Lattice filter to realize linear-phase two-channel filter bank. (© Springer Press [5])

addition are implemented. The design uses 420 LEs, no embedded multiplier, and has an $F_{max}=58.11$ MHz registered performance using the TimeQuest slow 85C model.

The VHDL simulation is shown in Fig. 5.66. The simulation shows the response to an impulse with amplitude 100 at even and odd positions for the filters $G(z)$ and $H(z)$, respectively.

5.24

If we compare the size of the lattice with the direct polyphase implementation of $G(z)$ shown in Example 5.1 on p. 310 (LEs multiplied by two), we note that both designs have about the same size. Although the lattice implementation needs only five multipliers, compared with eight multipliers for the polyphase implementation, we note that in the polyphase implementation we can use the RAG technique to implement the coefficients of the transposed filter, while in the lattice we must implement single multipliers, which, in general, are less efficient.

Linear-phase two-channel filter bank. We have already seen in Chap. 3 that if a linear filter has even or odd symmetry, 50% of multiplier resources can be saved. The same symmetry also applies for polyphase decomposition of the filters if the filters have even length. In addition, these filters may run at twice the speed.

If $G(z)$ and $H(z)$ have the same length, another implementation using lattice filters can further decrease the implementation effort, as shown in Fig. 5.67. Notice that the lattice is different from the lattice used for the orthogonal filter bank shown in Fig. 5.65.

The following example demonstrates how to convert a direct architecture into a lattice filter.

Example 5.25: Lattice for $L = 4$ Linear-Phase Filter

One filter configuration in Example 5.21 (p. 384) was a linear-phase filter pair, with both filters of length 4. The filters are

$$G(z) = \frac{1}{4} (1 + 3z^{-1} + 3z^{-2} + 1z^{-3}) \quad (5.100)$$

and

$$H(z) = \frac{1}{4} (-1 + 3z^{-1} + 3z^{-2} - 1z^{-3}). \quad (5.101)$$

The transfer functions for the two-channel length-4 linear-phase lattice filters are:

Table 5.6. Effort to compute two-channel filter banks if both filter are of length L

Type	Number of real multipliers	Number of real adders	see Fig.	Speed	Can use RAG ?
Polyphase with any coefficients					
Direct FIR filtering	$2L$	$2L - 2$	5.60	$2f_s$	✓
Lifting	$\approx L$	$\approx L$	5.62	$2f_s$	—
Quadrature mirror filter (QMF)					
Identical polyphase filter	L	L	5.63	$2f_s$	✓
Orthogonal filter					
Transposed FIR filter	L	$2L - 2$	5.64	f_s	✓
Lattice	$L + 1$	$3L/4$	5.65	$2f_s$	—
Linear-phase filter					
Symmetric filter	L	$2L - 2$	3.5	$2f_s$	✓
Lattice	$L/2$	$3L/2 - 1$	5.67	$2f_s$	—

$$G(z) = ((1 + a[0]) + a[0]z^{-1} + a[0]z^{-2} + (1 + a[0])z^{-3}) s \quad (5.102)$$

$$H(z) = (-(1 + a[0]) + a[0]z^{-1} + a[0]z^{-2} - (1 + a[0])z^{-3}) s. \quad (5.103)$$

Comparing (5.100) with (5.102), we find

$$s = -1/2 \quad a[0] = -1.5. \quad (5.104)$$

5.25

Note that, compared with the direct implementation, only about one quarter of the multipliers are required.

The disadvantage of the linear-phase lattice is that not all linear-phase filters can be implemented. Specifically, $G(z)$ must be even symmetric, $H(z)$ must be odd symmetric, and both filters must be of the same length, with an even number of samples.

Comparison of implementation options. Finally, Table 5.6 compares the different implementation options, which include the general case and special types like QMF, linear-phase and orthogonal.

Table 5.6 shows the required number of multipliers and adders, the reference figure, the maximum input rate, and the structurally important question of whether the coefficients can be implemented using reduced adder graph technique, or occur as single-multiplier coefficients. For shorter filters, the lattice structure seems to be attractive, while for longer filters, RAG will most often produce smaller and faster designs. Note that the number of multipliers and adders in Table 5.6 are an estimate of the hardware effort required for

the filter, and *not* the typical number found in the literature for the computational effort per input sample in a PDSP/ μ P solution [125, 162].

Excellent additional literature about two-channel filter banks is available (see [123, 159, 162, 163]).

5.8 Wavelets

A time-frequency representation of signals processed through transform methods has proven beneficial for audio and image processing [159, 164, 165]. Many signals subject to analysis are known to have statistically constant properties for only short time frames (e.g., speech or audio signals). It is therefore reasonable to analyze such signals in a short window, compute the signal parameter, and slide the window forward to analyze the next frame. If this analysis is based on Fourier transforms, it is called a *short-term Fourier transform* (STFT).

A short-term Fourier transform (STFT) is formally defined by

$$X(\tau, f) = \int_{-\infty}^{\infty} x(t) w(t - \tau) e^{-j2\pi ft} dt, \quad (5.105)$$

i.e., it slides a window function $w(t - \tau)$ over the signal $x(t)$, and produces a continuous time-frequency map. The window should taper smoothly to zero, both in frequency and time, to ensure localization in frequency Δ_f and time Δ_t of the mapping. One weight function, the Gaussian function ($g(t) = e^{-t^2}$), is optimal in this sense, and provides the minimum (Heisenberg principle) product $\Delta_f \Delta_t$ (i.e., best localization), as proposed by Gabor in 1949 [166]. The discretization of the Gabor transform leads to the discrete Gabor transform (DGT). The Gabor transform uses identical resolution windows throughout the time and frequency plane (see Fig. 5.69a). Every rectangle in Fig. 5.69a has exactly the same shape, but often a constant Q (i.e., the quotient of bandwidth to center frequency) is desirable, especially in audio and image processing. That is, for high frequencies we wish to have broadband filters and short sampling intervals, while for low frequencies, the bandwidth should be small and the intervals larger. This can be accomplished with the continuous wavelet transform (CWT), introduced by Grossmann and Morlet [167],

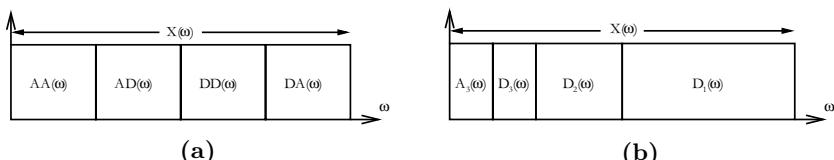


Fig. 5.68. Frequency distribution for (a) Fourier (constant bandwidth) and (b) constant Q

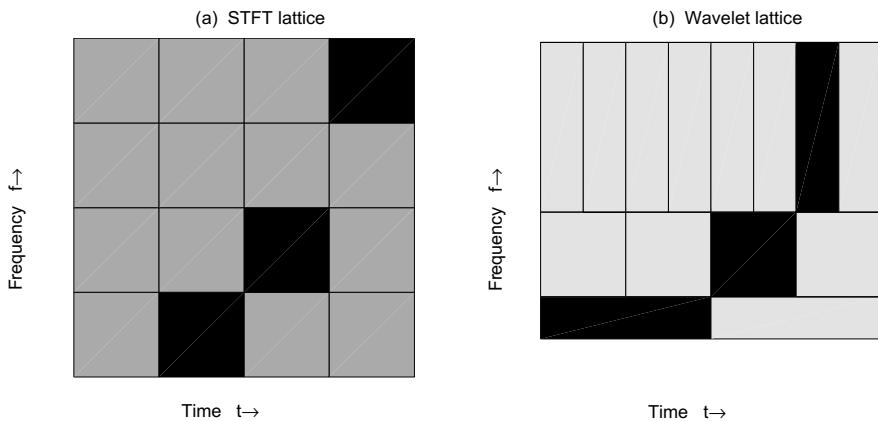


Fig. 5.69. Time frequency grids for a chirp signal. (a) Short-term Fourier transform. (b) Wavelet transform

$$\text{CWT}(\tau, f) = \int_{-\infty}^{\infty} x(t) h\left(\frac{t-\tau}{s}\right) dt, \quad (5.106)$$

where $h(t)$, known from the Heugens principle in physics, is called a small wave or *wavelet*. Some typical wavelets are displayed in Fig. 5.70.

If we use now as a wavelet

$$h(t) = \left(e^{j2\pi kt} - e^{-k^2/2} \right) e^{-t^2/2} \quad (5.107)$$

we still enjoy the “optimal” properties of the Gaussian window, but now with different scales in time and frequency. This so-called Morlet transform is also subject to quantization, and is then called the discrete Morlet transformation (DMT) [168]. In the discrete case the lattice points in time and frequency are shown in Fig. 5.69b. The exponential term $e^{-k^2/2}$ in (5.107) was introduced

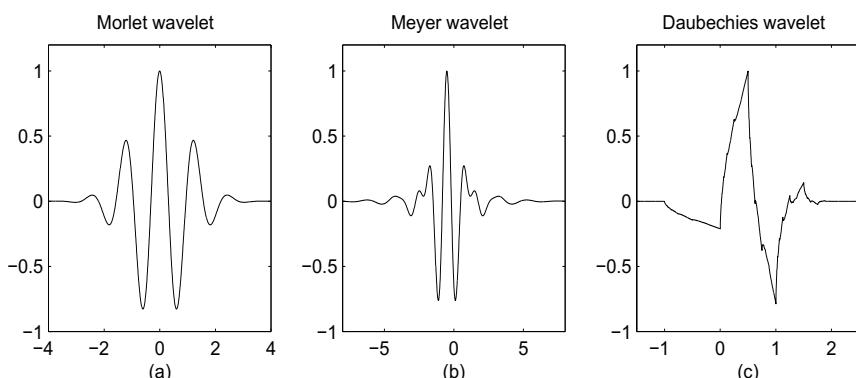


Fig. 5.70. Some typical wavelets from Morlet, Meyer, and Daubechies

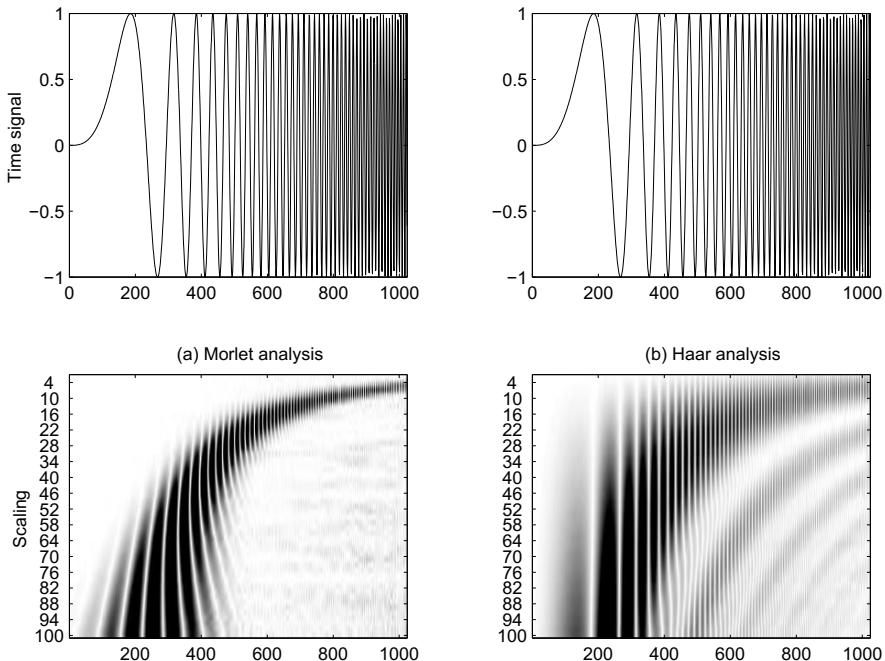


Fig. 5.71. Analysis of a chirp signal with (a) Discrete Morlet transform. (b) Haar transform

such that the wavelet is DC free. The following examples show the excellent performance of the Gaussian window.

Example 5.26: Analysis of a Chirp Signal

Figure 5.71 shows the analysis of a constant amplitude signal with increasing frequency. Such signals are called chirp signals. If we applied the Fourier transform we would get a uniform spectrum, because all frequencies are present. The Fourier spectrum does not preserve time-related information. If we use instead an STFT with a Gaussian window, i.e., the Morlet transform, as shown in Fig. 5.71a, we can clearly see the increasing frequency. The Gaussian window shows the best localization of all windows. On the other hand, with a Haar window we would have less computational effort, but, as can be seen from Fig. 5.71b, the Haar window will achieve less precise time-frequency localization of the signal.

5.26

Both DGT and DMT provide good localization by using a Gaussian window, but both are computationally intensive. An efficient multiplier-free implementation is based on two ideas [168]. First, the Gaussian window can be sufficiently approximated by a convolution of (≥ 3) rectangular functions, and second, single-passband frequency-sampling filters (FSF) can be

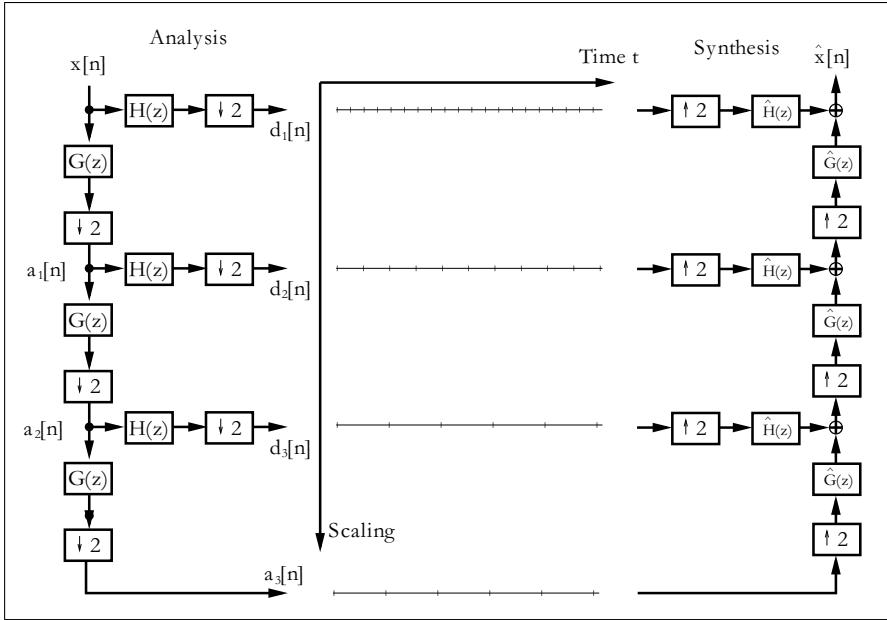


Fig. 5.72. Wavelets tree decomposition in three octaves. (© Springer Press [5])

efficiently implemented by defining algebraic integers over polynomial rings, as introduced in [168].

In the following, we wish to focus our attention on a newly popular analysis method called the *discrete wavelet transform*, which better exploits the auditory and visual human perception mode (i.e., constant Q), and also can often be more efficiently computed, using $\mathcal{O}(n)$ complexity algorithms.

5.8.1 The Discrete Wavelet Transformation

A discrete-time version of the analog model leads to the *discrete wavelet transform* (DWT). In practical applications, the DWT is restricted to the discrete time *dyadic DWT* with $a = 2$, and will be considered in the following. The DWT achieves the constant Q bandwidth distribution shown in Fig. 5.68b and Fig. 5.69b by always applying the two-channel filter bank in a filter tree to the lowpass signal, as shown in Fig. 5.72.

We now wish to focus on what conditions for the CWT wavelet allow it to be realized with a two-channel DWT filter bank. We may argue that if we sample a continuous wavelet at an appropriate rate (above the Nyquist rate), we may call the sampled version a DWT. But, in general, only those continuous wavelet transforms that can be realized with a two-channel filter bank are called DWT.

Closely related to whether a continuous wavelet $\psi(t)$ can be realized with a two-channel DWT, is the question of whether the *scaling equation*

$$\phi(t) = \sum_n g[n] \phi(2t - n) \quad (5.108)$$

exists, where the actual wavelet is computed with

$$\psi(t) = \sum_n h[n] \phi(2t - n), \quad (5.109)$$

where $g[n]$ is a lowpass, and $h[n]$ a highpass filter. Note that $\phi(t)$ and $\psi(t)$ are continuous functions, while $g[n]$ and $h[n]$ are sample sequences (but still may also be IIR filters). Note that (5.108) is similar to the *self-similarity* ($\phi(t) = \phi(at)$) exhibited by fractals. In fact, the scaling equation may iterate to a fractal, but that is, in general, not the desired case, because most often a smooth wavelet is desired. The smoothness can be improved if we use a filter with maximal numbers of zeros at π .

We consider now backwards reconstruction: we start with the filter $g[n]$, and construct the corresponding wavelet. This is the most common case, especially if we use the half-band design from Algorithm 5.20 (p. 384) to generate perfect reconstruction filter pairs of the desired length and property.

To get a graphical interpretation of the wavelet, we start with a rectangular function (box function) and build, according to (5.108), the following graphical iteration:

$$\phi^{(k+1)}(t) = \sum_n g[n] \phi^{(k)}(2t - n). \quad (5.110)$$

If this converges to a stable $\phi(t)$, the (new) wavelet is found. This iteration obviously converges for the Haar filter $\{1, 1\}$ immediately after the first iteration, because the sum of two box functions scaled and added is again a box function, i.e.,

$$\begin{array}{c} r(t) \\ \hline \end{array} = \begin{array}{c} r(2t) + r(2t-1) \\ \hline \end{array}$$

Let us now graphically construct the wavelet that belongs to the filter $g[n] = \{1, 1, 1, 1\}$, which we will call Hutlet4 [169].

Example 5.27: Hutlet of Length-4

We start with four box functions weighted by $g[n] = \{1, 1, 1, 1\}$. The sum shown in Fig. 5.73a is the starting $\phi^{(1)}(t)$. This function is scaled by two, and the sum gives a two-step function. After 10 iterations we already get a very smooth trapezoid function. If we now use the QMF relation, from (5.78) (p. 380), to construct the actual wavelet, we get the Hutlet4, which has two triangles as shown in Fig. 5.74.

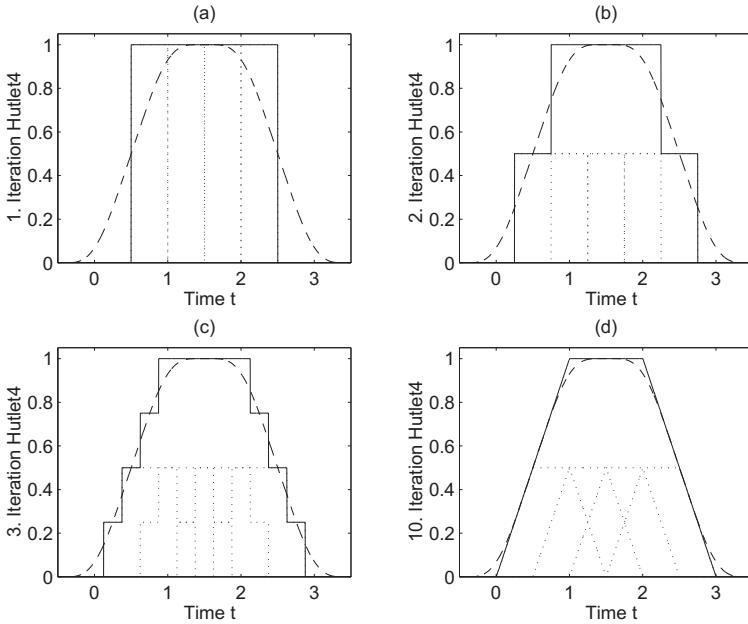


Fig. 5.73. Iteration steps 1, 2, 3, and 10 for Hutlet4. (solidline) $\phi^{(k+1)}(t)$; (dottedline) $\phi^{(k)}(2t - n)$; and ideal Hut-function (dashed)

We note that $g[n]$ is the impulse response of the moving-average filter, and can be implemented as an one-stage CIC filter [170]. Figure 5.74 shows all scaling functions and wavelets for this type of wavelet with even length coefficients.

As noted before, the iteration defined by (5.110) may also converge to a fractal. Such an example is shown in Fig. 5.75, which is the wavelet for the length-5 “moving average filter.” This indicates the challenge of the filter selection $g[n]$: it may converge to a smooth or, totally chaotic function, depending only on an apparently insignificant property like the length of the filter!

We still have not explained why the two-scale equation (5.108) is so important for the DWT. This can be better understood if we rearrange the downampler (compressor) and filter in the analysis part of the DWT, using the “Noble” relation

$$(\downarrow M) H(z) = H(z^M) (\downarrow M), \quad (5.111)$$

which was introduced in Sect. 5.1.1, p. 306. The results for a three-level filter bank are shown in Fig. 5.76. If we compute the impulse response of the cascade sequences, i.e.,

$$H(z) \leftrightarrow d_1[k/2]$$

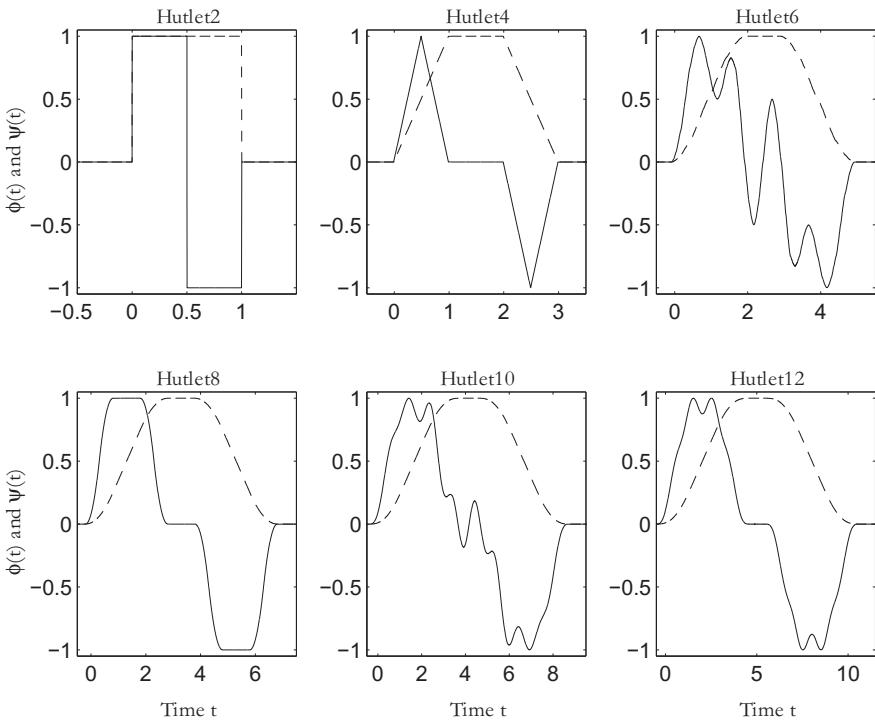


Fig. 5.74. The Hutlet wavelet family (solid line) and scaling function (dashed line) after 10 iterations. (© Springer Press [5])

$$\begin{aligned}
 G(z)H(z^2) &\leftrightarrow d_2[k/4] \\
 G(z)G(z^2)H(z^4) &\leftrightarrow d_3[k/8] \\
 G(z)G(z^2)G(z^4) &\leftrightarrow a_3[k/8],
 \end{aligned}$$

we find that a_3 is an approximation to the scaling function, while d_3 gives an approximation to the mother wavelet, if we compare the graphs with the continuous wavelet shown in Fig. 5.70 (p. 396).

This is not always possible. For instance, for the Morlet wavelet shown in Fig. 5.70 (p. 396), no scaling function can be found, and a realization using the DWT is *not* possible.

Two-channel DWT design examples for the Daubechies length-4 filter have already been discussed, in combination with polyphase representation (Example 5.1, p. 310), and the lattice implementation of orthogonal filters in Example 5.24 (p. 390).

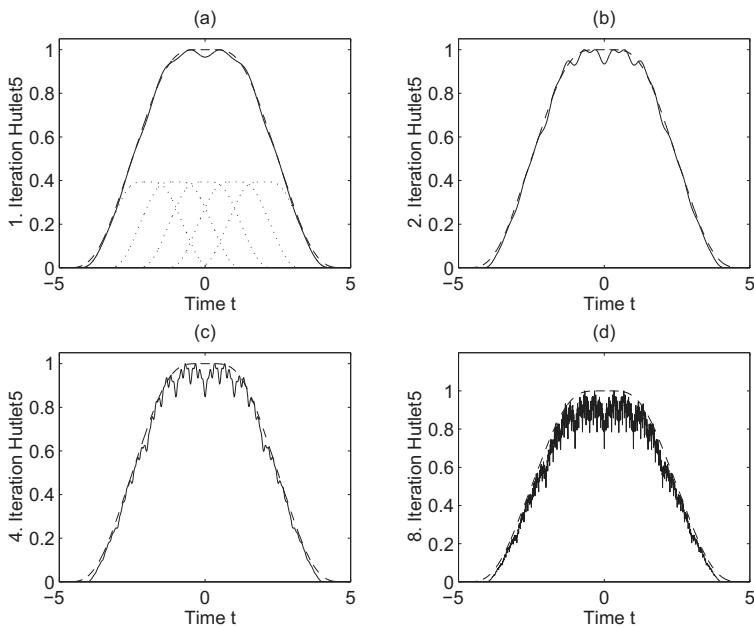


Fig. 5.75. Iteration step 1, 2, 4, and 8 for Hutlet5. The sequence converges to a fractal

5.8.2 Discrete Wavelet Transformation Applications

When wavelets became popular in the 1980s many researcher were very excited and thought that finally a tool even more useful than FFT had arrived. At the time Wim Swelden's electronic newsletter called "Wavelet Digest" (see www.wavelet.org) had over 20,000 subscribers that wanted to know the latest results in the wavelet community. After some time, reality settled in and many realized that wavelets indeed can be superior to Fourier based methods for some applications, or open up new applications, but FFT on the other hand supports fast convolution, fast correlation or spectrum estimation and these applications are *not* supported by DWT as one can easily verify. However, there are a couple of applications when DWT is superior and we want to discuss the most important in this section. These applications are:

- 1) Image compression by avoiding blocking effects of DCT method, e.g., JPEG↔JPEG2000, see Chap. 10.
- 2) De-noising of signals and preserving the shape and nature of the signals.
- 3) Image enhancement.
- 4) Detection and classification of discontinuities.

The image compression features are discussed in Chap. 10. Image enhancements and detection of discontinuities are less frequently used and are

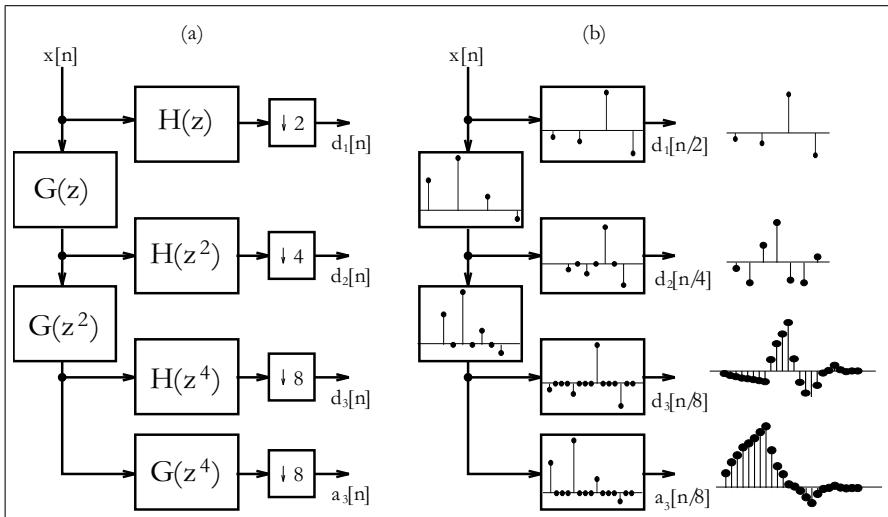


Fig. 5.76. DWT filter bank rearrange using Noble relations. (a) Transfer function in the z -domain. (b) Impulse response for the length-4 Daubechies filters

explained in the literature; see for instance [5]. Let us in the following discuss the second most popular application: the de-noising methods. For a de-noising in an FFT-based scheme we would transform our signal in the Fourier domain and set small Fourier coefficients to zero in the hope that these small components are indeed the noise in the signal and not signal components; see Fig. 5.77. In the DWT method we would carry out similar processing. We would for instance use a three level analysis of the signal, then set small wavelet coefficient to zero and reconstruct the signal using the synthesis steps. This method is shown in Fig. 5.78 for a Haar filter bank which we will implement below. We note two modification to the standard tree scheme from Fig. 5.72, p. 398. Since we use causal filters additional delays are used in the reconstruction a.k.a. synthesis tree to put the signals in phase. The second modification to de-noising using the MATLAB function `wden` is that we use integer filters and not the orthonormal Haar filters where all coefficients have a $1/\sqrt{2}$ scale factor that would produce much higher hardware costs. Such a SIMULINK model also allows us to generate test data that we need later in the VHDL design. Impulse response will require many test sequences since the systems has many up- and down-samplers. For a triangular input the “detail” signal d_k will not change much. The test data that is most beneficial is therefore a quadratic type of input signal. We can generate it with a convolution of three box functions. Figure 5.79a shows the analysis data and Fig. 5.79b the delayed match in the signals $a_3(t) = s_3(t - \tau)$ and $a_2(t) = s_2(t - \tau)$ in the synthesis signals. These signals should be used in the test and debug phase of the DWT de-noising HDL code.

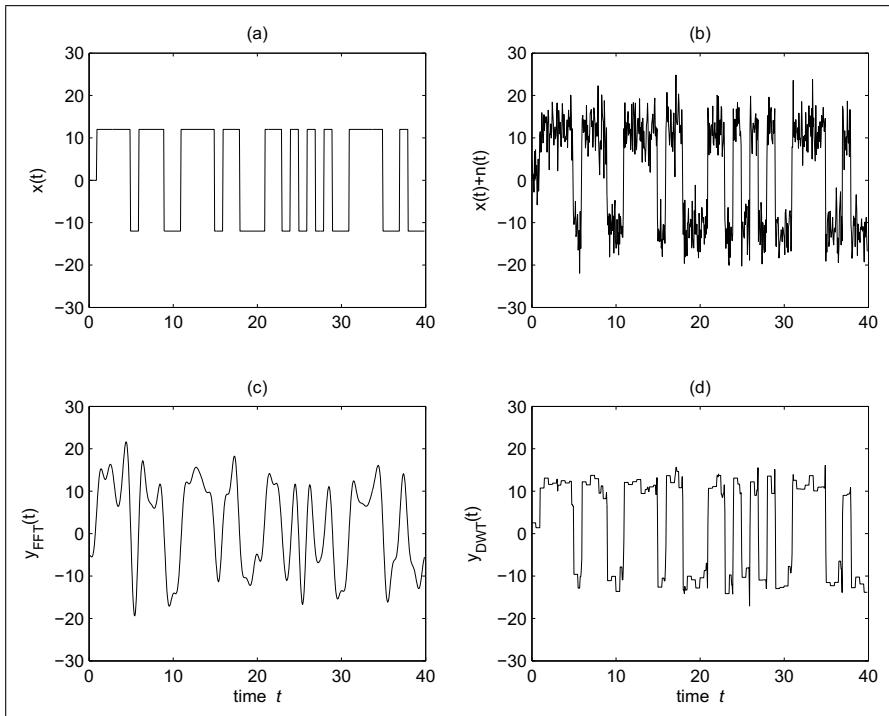


Fig. 5.77. Comparison for de-noising method using DFT/FFT and Haar DWT. (a) Original signal. (b) Original with added noise. (c) De-noising results using FFT. (d) De-noising using DWT

After we have the basic system running we can test the de-noising with a more sophisticated signal. Figure 5.77 shows a comparison of DWT and FFT-based methods. We use a pseudo random sequence that can be found in the phase component of DCF77 radio controlled watch signals that is used for synchronization purpose. Figure 5.77a shows for each bit 16 samples and Fig. 5.77b shows the signal with the additional noise. If we compare the results in Fig. 5.77c,d we see that the DWT much better preserves the shape of the input signal and subsequent circuits such as a PLL will be much easier to synchronize. Setting small values in the FFT to zero is nothing else than a lowpass filtering and we should expect that sharp edges will no longer be present in the output signal of the FFT de-noising. We also see from this example that the wavelet in use should be a good representation of the input signal (without noise). The last item to discuss is the selection of the threshold. The immediately apparent method would be to use a certain percentage, e.g., 10% or 25% of the maximum wavelet coefficient at that octave. More sophisticated methods will first compute the variance of the signal and then for high variance we would assume that the information

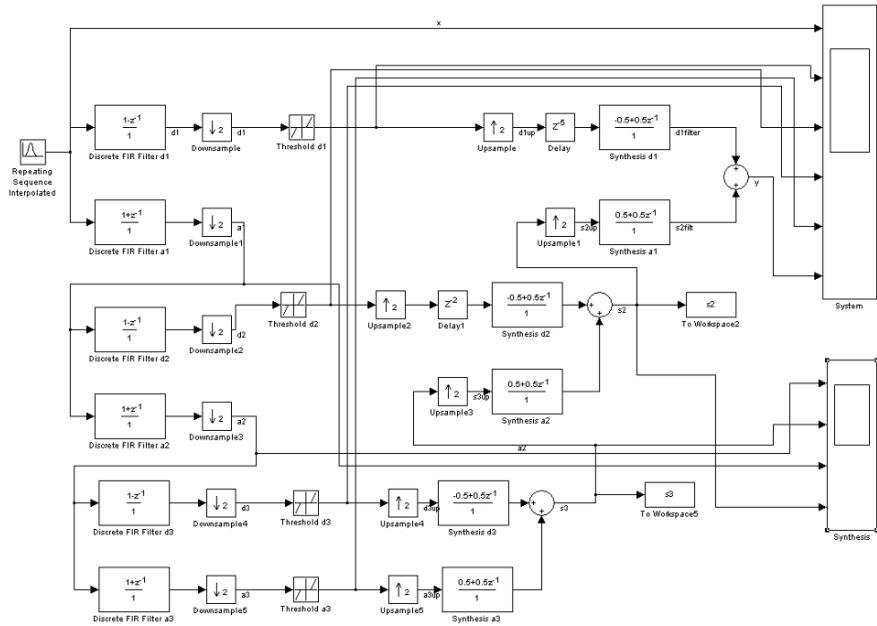


Fig. 5.78. A three level wavelet denoising scheme

content in this octave is large and we use a small threshold value. In the MATLAB toolbox we find four different methods to compute the threshold. One particularly interesting observation was made by Donoho and Johnstone [171] who found for Gaussian noise a threshold

$$T = \sigma \sqrt{2 \log_e(N)} \quad (5.112)$$

should be chosen, where σ is the standard deviation of the Gaussian white noise and N the number of samples. The threshold increases with N due to the fact that for large data sets the probability of large amplitudes in Gaussian noise also increases. However, the $\sqrt{\log_e(N)}$ function increase very slowly. If we increase the sample size from $N = 512$ to $N = 1024$ then T changes by only 5% [171, 172]. The larger problem with the threshold (5.112) is that we need a good Gaussian noise estimation. To this end it has been suggested [173] that we use the median of the first wavelet analysis, i.e., d_1 signal and estimate

$$\hat{\sigma} = \frac{1}{0.6745} \text{ Median}(d_1[m])|_{0 \leq m \leq N/2} \quad (5.113)$$

Now we have the task of computing the median of the full data set. To compute a median of 1K or more data will be a quite challenging hardware design since we need to sort these 1K data. Using a less complicated threshold estimation still provides excellent results as the following hardware implementation shows.

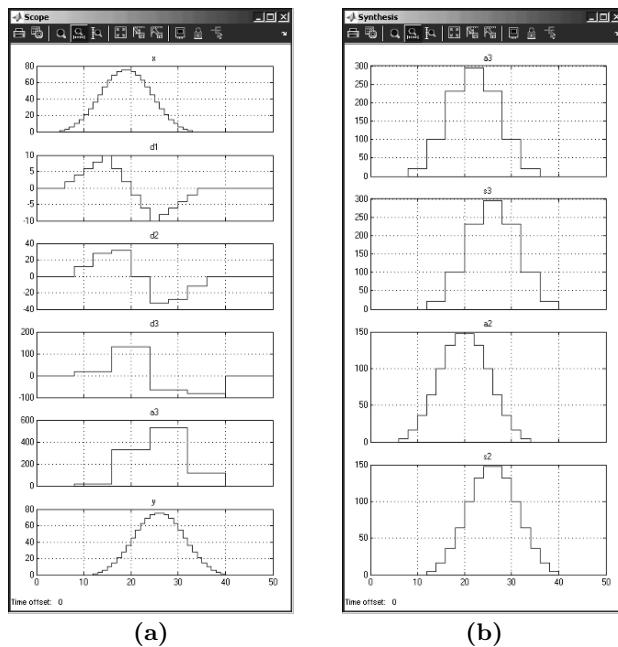


Fig. 5.79. SIMULINK DWT test bench data. (a) Analysis test bench data. (b) Synthesis data

Example 5.28: DWT De-noising of Digital Signal

The following VHDL code¹⁶ shows the three level DWT de-noising:

```

PACKAGE n_bits_int IS           -- User defined types
  SUBTYPE U3 IS INTEGER RANGE 0 TO 7;
  SUBTYPE S16 IS INTEGER RANGE -2**15 TO 2**15-1;
  TYPE A0_11S16 IS ARRAY (0 TO 11) OF S16;
  TYPE A0_29S16 IS ARRAY (0 TO 29) OF S16;
END n_bits_int;

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-----> Interface
ENTITY dwtden IS
  GENERIC (D1L : INTEGER := 28; -- D1 buffer length
           D2L : INTEGER := 10); -- D2 buffer length
  PORT (clk      : IN STD_LOGIC; -- System clock
        
```

¹⁶ The equivalent Verilog code `dwt�en.v` for this example can be found in Appendix A on page 847. Synthesis results are shown in Appendix B on page 881.

```

reset      : IN STD_LOGIC; -- Asynchron reset
x_in       : IN S16;    -- System input
t4d1, t4d2, t4d3, t4a3 : IN S16;   -- Threshold
d1_out     : OUT S16;   -- Level 1 detail
a1_out     : OUT S16;   -- Level 1 approximation
d2_out     : OUT S16;   -- Level 2 detail
a2_out     : OUT S16;   -- Level 2 approximation
d3_out     : OUT S16;   -- Level 3 detail
a3_out     : OUT S16;   -- Level 3 approximation
s3_out, a3up_out, d3up_out : OUT S16; -- L3 debug
s2_out, s3up_out, d2up_out : OUT S16; -- L2 debug
s1_out, s2up_out, d1up_out : OUT S16; -- L1 debug
y_out      : OUT S16); -- System output
END dwtden;
-----
ARCHITECTURE fpga OF dwtden IS

SIGNAL count  : U3; -- Cycle 2**max level
SIGNAL x, xd  : S16; -- Input delays
SIGNAL a1, d1, a2, d2, a3, d3 : S16; -- Analysis filter
SIGNAL d1t, d2t, d3t, a3t : S16; -- Before thresholding
SIGNAL a1up, a3up, d3up : S16;
SIGNAL a1upd, s3upd, a3upd, d3upd : S16;
SIGNAL a1d, a2d : S16; -- Delay filter output
SIGNAL ena1, ena2, ena3 : BOOLEAN; -- Clock enables
SIGNAL t1, t2, t3 : STD_LOGIC; -- Toggle flip-flops
SIGNAL s2, s3up, s3, d2syn : S16;
SIGNAL s1, s2up, s2upd : S16;
-- Delay lines for d1 and d2
SIGNAL d2upd : A0_11S16;
SIGNAL diupd : A0_29S16;
BEGIN
  FSM: PROCESS (reset, clk)      -----> Control the system
BEGIN
  IF reset = '1' THEN           -- sample at clk rate
    count <= 0;                -- Asynchronous reset
  ELSIF rising_edge(clk) THEN
    IF count = 7 THEN
      count <= 0;
    ELSE
      count <= count + 1;
    END IF;
  CASE count IS -- Level 1 enable
    WHEN 1 | 3 | 5 | 7 =>
      ena1 <= TRUE;
    WHEN others =>
      ena1 <= FALSE;
  END CASE;
  CASE count IS -- Level 2 enable
    WHEN 1 | 5 =>
      ena2 <= TRUE;
    WHEN others =>

```

```

    ena2 <= FALSE;
END CASE;
CASE count IS -- Level 3 enable
WHEN 5 =>
    ena3 <= TRUE;
WHEN others =>
    ena3 <= FALSE;
END CASE;
END IF;
END PROCESS FSM;

-- Haar analysis filter bank
Analysis: PROCESS(clk, reset)      -----> Behavioral Style
BEGIN
IF reset = '1' THEN                -- Asynchronous clear
    x <= 0; xd <= 0;
    d1t <= 0; a1 <= 0; a1d <= 0;
    d2t <= 0; a2 <= 0; a2d <= 0;
    d3t <= 0; a3t <= 0;
ELSIF rising_edge(clk) THEN
    x <= x_in;
    xd <= x;
    IF ena1 THEN -- Level 1 analysis
        d1t <= x - xd;
        a1 <= x + xd;
        a1d <= a1;
    END IF;
    IF ena2 THEN -- Level 2 analysis
        d2t <= a1 - a1d;
        a2 <= a1 + a1d;
        a2d <= a2;
    END IF;
    IF ena3 THEN -- Level 3 analysis
        d3t <= a2 - a2d;
        a3t <= a2 + a2d;
    END IF;
END IF;
END PROCESS;

-- Thresholding of d1, d2, d3 and a3
d1 <= d1t WHEN abs(d1t) > t4d1 ELSE 0;
d2 <= d2t WHEN abs(d2t) > t4d2 ELSE 0;
d3 <= d3t WHEN abs(d3t) > t4d3 ELSE 0;
a3 <= a3t WHEN abs(a3t) > t4a3 ELSE 0;

-- Down followed by up sampling is implemented by setting
-- every 2. value to zero
Synthesis: PROCESS(clk, reset)      -----> Behavioral Style
BEGIN
IF reset = '1' THEN                -- Asynchronous clear
    t1 <= '0'; t2 <= '0'; t3 <= '0';
    s3up <= 0;s3upd <= 0;
    d3up <= 0; a3up <= 0; a3upd<=0; d3upd <= 0;

```

```

s3 <= 0; s2 <= 0;
s1 <= 0; s2up <= 0; s2upd <= 0;
FOR k IN 0 TO D2L+1 LOOP -- Clear array match s3up
    d2upd(k) <= 0;
END LOOP;
FOR k IN 0 TO D1L+1 LOOP -- Clear array match s2up
    d1upd(k) <= 0;
END LOOP;
ELSIF rising_edge(clk) THEN
    t1 <= NOT t1; -- toggle FF level 1
    IF t1 = '1' THEN
        d1upd(0) <= d1;
        s2up <= s2;
    ELSE
        d1upd(0) <= 0;
        s2up <= 0;
    END IF;
    s2upd <= s2up;
    FOR k IN 1 TO D1L+1 LOOP -- Delay to match s2up
        d1upd(k) <= d1upd(k-1);
    END LOOP;
    s1 <= (s2up + s2upd - d1upd(D1L) + d1upd(D1L+1))/2;

    IF ena1 THEN
        t2 <= NOT t2; -- toggle FF level 2
        IF t2 = '1' THEN
            d2upd(0) <= d2;
            s3up <= s3;
        ELSE
            d2upd(0) <= 0;
            s3up <= 0;
        END IF;
        s3upd <= s3up;
        FOR k IN 1 TO D2L+1 LOOP -- delay to match s3up
            d2upd(k) <= d2upd(k-1);
        END LOOP;
        s2 <= (s3up + s3upd - d2upd(D2L) + d2upd(D2L+1))/2;
    END IF;

    IF ena2 THEN -- Synthesis level 3
        t3 <= NOT t3; -- toggle FF
        IF t3='1' THEN
            d3up <= d3;
            a3up <= a3;
        ELSE
            d3up <= 0;
            a3up <= 0;
        END IF;
        a3upd <= a3up;
        d3upd <= d3up;
        s3 <= (a3up + a3upd - d3up + d3upd)/2;
    END IF;
END IF;

```

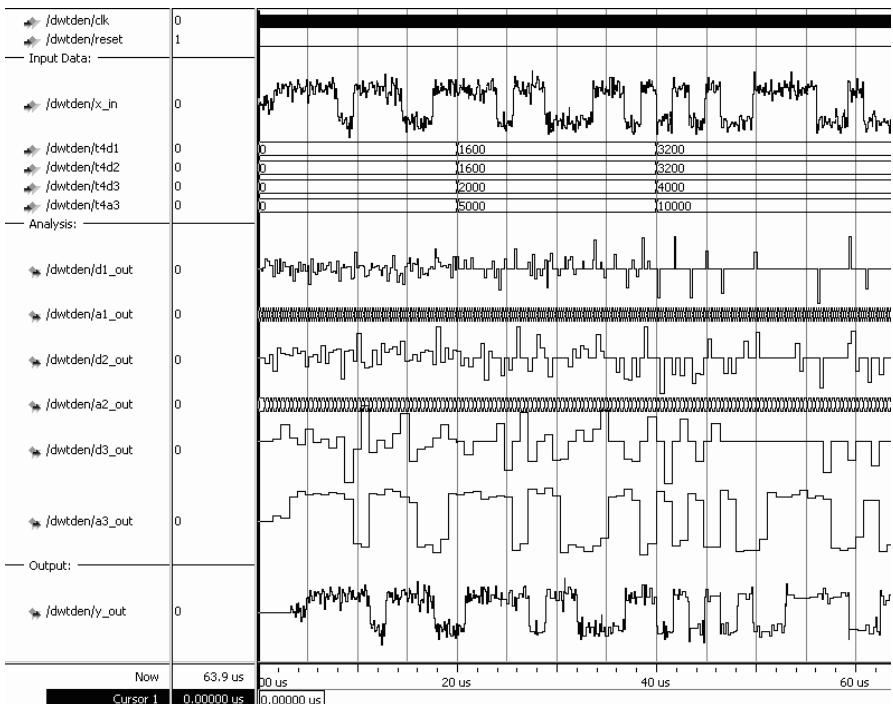


Fig. 5.80. VHDL simulation of the DWT de-noising

```

END PROCESS;

a1_out <= a1; -- Provide some test signal as outputs
d1_out <= d1;
a2_out <= a2;
d2_out <= d2;
a3_out <= a3;
d3_out <= d3;
a3up_out <= a3up;
d3up_out <= d3up;
s3_out <= s3;
s3up_out <= s3up;
d2up_out <= d2upd(D2L);
s2_out <= s2;
s1_out <= s1;
s2up_out <= s2up;
d1up_out <= d1upd(D1L);
y_out <= s1;

```

```

END fpga;
```

After the coding for the data type used and the ENTITY the ARCHITECTURE of the design follows. The first PROCESS is the FSM, which includes the control flow and generation of the enable signals for the analysis and syn-

thesis filters. The full round takes eight clock cycles. The next **Analysis: PROCESS** blocks includes the three pairs of analysis filters controlled by the three enable signals. Then four concurrent statements for the threshold follow that set small wavelet coefficients to zero depending on the four “threshold for d_k ” input port signals. The last **Synthesis: PROCESS** block reconstructs the input signals in the three levels. Note that substantially longer delays are required to put the signals into synchronization than in the SIMULINK simulation in Fig. 5.78 since up- and downsampling are realized with synchronous registers. The design uses 879 LEs, no embedded multiplier, and has an **Fmax**=120.93 MHz registered performance using the TimeQuest slow 85C model.

A simulation of the filter is shown in Fig. 5.80. The simulation first shows the control and threshold signals of the FSM. The DCF77 40-bit sequence is used for **x_in**. The signals $s(k)$ have been generated in MATLAB and then printed to file for the MODELTECH simulation with

```
for k=1:L
    fprintf(fid,'force x_in %d %dns\r\n',round(256*s(k)),...
           100*k);
end
```

The floating point values have been scaled by 256 to provide 8-bit fractional precision in the MODELSIM HDL simulation. Due to the large amount of data not all test port signals are shown. The signals of interest have been displayed as (analog) waveform by selection of the **Format** → **Analog (automatic)** display for **x**, **d1**, **d2**, **d3**, **a3**, and **y**. The threshold values have been increased from zero (no threshold) to 25% and 50% of the maximum wavelet coefficient observed. Note in particular how well the 50% signals remove the noise in **y_out** and at the same time preserve the shape of the input digital signal.

5.28

Exercises

Note: If you have no prior experience with the Quartus II software, refer to the case study found in Sect. 1.4.3, p. 32. If not otherwise noted use the EP4CE115F29C7 from the Cyclone IV E family for the Quartus II synthesis evaluations.

5.1: Let $F(z) = 1 + z^{-d}$. For which d do we have a half-band filter according to Definition 5.7 (p. 339)?

5.2: Let $F(z) = 1 + z^{-5}$ be a half-band filter.

- (a) Draw $|F(\omega)|$. What kind of symmetry does this filter have?
- (b) Use Algorithm 5.20 (p. 384) to compute a perfect-reconstruction real filter bank. What is the total delay of the filter bank?

5.3: Use the half-band filter F3 from Example 5.21 (p. 384) to build a perfect-reconstruction filter bank, using Algorithm 5.20 (p. 384), of length

- (a) 1/7.
- (b) 2/6.

5.4: How many different filter pairs can be built, using F3 from Example 5.21 (p. 384), if both filters are

- (a) Complex.
- (b) Real.
- (c) Linear-phase.
- (d) Orthogonal filter bank.

5.5: Use the half-band filter $F_2(z) = 1 + 2z^{-1} + z^{-2}$ to compute, based on Algorithm 5.20 (p. 384), all possible perfect-reconstruction filter banks.

5.6: (a) Compute the number of real additions and multiplications for a direct implementation of the critically sampled uniform DFT filter bank shown in Fig. 5.52 (p. 375). Assume the length L analysis and synthesis filters have complex coefficients, and the inputs are real valued.

(b) Assume an FFT algorithm is used that needs $(15N \log_2(N))$ real additions and multiplications. Compute the total effort for a uniform DFT filter bank, using the polyphase representation from Figs. 5.54 (p. 377) and 5.55 (p. 378), for R of length L complex filters.

(c) Using the results from (a) and (b) compute the effort for a critically sampled DFT filter bank with $L = 64$ and $R = 16$.

5.7: Use the lossy integrator from Example 5.15 (p. 377) to implement an $R = 4$ uniform DFT filter bank.

(a) Compute the analysis polyphase filter $H_k(z)$.

(b) Determine the synthesis filter $F_k(z)$ for perfect reconstruction.

(c) Determine the 4×4 DFT matrix. How many real additions and multiplications are used to compute the DFT?

(d) Compute the total computational effort of the whole filter bank, in terms of real additions and multiplications per input sample.

5.8: Analyze the frequency response of each Goodman and Carey half-band filter from Table 5.3 (p. 339). Zoom in on the passband to estimate the ripple of the filter.

5.9: Prove the perfect reconstruction for the *lifting* and *dual-lifting* scheme from (5.89) and (5.90) on p. 387.

5.10: (a) Implement the Daubechies length-4 filter using the lifting scheme from Example 5.23 (p. 387), with 8-bit input and coefficient, and 10-bit output quantization.

(b) Simulate the design with two impulses of amplitude 100, similar to Fig. 5.66 (p. 392).

(c) Determine the registered performance **Fmax** using the **TimeQuest** slow 85C model and the used resources (LEs, multipliers, and M9Ks).

(d) Compare the lifting design with the direct polyphase implementation (Example 5.1, p. 310) and with the lattice implementation (Example 5.24, p. 390), in terms of size and speed.

5.11: Use component instantiation of the two designs from Example 5.4 (p. 321) and Example 5.6 (p. 330) to compute the difference of the two filter outputs. Determine the maximum positive and negative deviation.

5.12: (a) Use the reduced adder graph design from Fig. 3.11 (p. 200) to build a half-band filter F6 (see Table 5.3, p. 339) for 8-bit inputs using Quartus II. Use the transposed FIR structure (Fig. 3.3, p. 181) as the filter architecture.

- (b) Verify the function via a simulation of the impulse response.
 (c) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks), of the F6 design.

5.13: (a) Compute the polyphase representation for F6 from Table 5.3, p. 339.
 (b) Implement the polyphase filter F6 with decimation $R = 2$ for 8-bit inputs with Quartus II.
 (c) Verify the function via a simulation of the impulse (one at even and one at odd) response.
 (d) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) of the polyphase design.
 (e) What are the advantages and disadvantages of the polyphase design, when compared with the direct implementation from Exercise 5.12 (p. 412), in terms of size and speed.

5.14: (a) Compute the 8-bit quantized DB4 filters $G(z)$ by multiplication of (5.95) with 256 and taking the integer part. Use the programm `csd3e.exe` from the CD-ROM or the data from Table 2.3, p. 68.

- (b1) Design the filter $G(z)$ only from Fig. 5.64, p. 389 for 9-bit inputs with Quartus II. Assume that input and coefficient are signed, i.e., only one additional guard bit is required for a filter of length 4.

(b2) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) of the filter $G(z)$.

(b3) What are the advantages and disadvantages of the CSD design, when compared with the programmable FIR filter from Example 3.1 (p. 182), in terms of size and speed.

(c1) Design the filter bank with $H(z)$ and $G(z)$ from Fig. 5.64, p. 389.

(c2) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) of the filter bank.

(c3) What are the advantages and disadvantages of the CSD filter bank design, when compared with the lattice design from Example 5.24, (p. 390), in terms of size and speed.

5.15: (a) Use the MAG coding from Table 2.3 (p. 68) to build the sinc filter from Example 5.12, (p. 350).

(a1) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) of the MAG `rc_sinc` design.

(b) Implement a pipelined adder tree to improve the throughput of the filter `rc_sinc` design.

(b1) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) for the improved design.

5.16: (a) Use the sinc filter data from Example 5.12, (p. 350) to estimate the implementation effort for an $R = 147/160$ rate changer. Assume that the two filters each account for 50% of the resources.

(b) Use the Farrow filter data from Example 5.13, (p. 358) to estimate the implementation effort for an $R = 147/160$ rate changer.

(c) Compare the two design options from (a) and (b) for small and large values of R_1 in terms of required LEs and registered performance F_{max} using the TimeQuest slow 85C model.

5.17: The Farrow combiner from Example 5.13, (p. 358) uses several multiplications in series. Pipeline register for the data and multiplier can be added to perform a

maximum-speed design.

(a) How many pipeline stages (total) are required for a maximum-speed design if we use:

(a1) an embedded array multiplier?

(a2) an LE-based multiplier?

(b) Design the HDL code for a maximum-speed design using:

(b1) an embedded array multiplier

(b2) an LE-based multiplier

(c) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) for the improved designs.

5.18: (a) Compute and plot using MATLAB or C the impulse response of the IIR filter given in (5.56), p. 363. Is this filter stable?

(b) Simulate the IIR filter using the following input signal: $x = [0, 0, 0, 1, 2, 3, 0, 0, 0]$; using the filter $F = [1 \ 4 \ 1]/6$ determine

(b1) x filtered by $F(z)$ followed by $1/F(z)$ and plot the results.

(b2) x filtered by $1/F(z)$ followed by $F(z)$ and plot the results.

(c) Split the filter $1/F(z)$ in a stable causal and a-causal part and apply the causal first-to-last sample while the a-causal is applied last-to-first sample. Repeat the simulation in (b).

5.19: (a) Plot the filter transfer function of the IIR filter given in (5.56), p. 363.

(b) Build the length-11 IFFT of the filter and apply a DC correction, i.e., $\sum_k h(k) = 0$.

(c) Unser et al. [149] determined the following length-11 FIR approximation for the IIR: $[-0.0019876, 0.00883099, -0.0332243, 0.124384, -0.46405, 1.73209, -0.46405, 0.124384, -0.0332243, 0.00883099, -0.0019876]$. Plot the impulse response and transfer function of this filter.

(d) Determine the error of the two solutions in (b) and (c) by computing the convolution with the filter (5.55) and building the square sum of the elements (excluding 1).

5.20: Use the Farrow equation (5.60) (p. 366) for B-splines to determine the Farrow matrix for the c_k for

(a) I-MOMS with $\phi(t) = \beta^3(t) - \frac{1}{6} \frac{d^2\beta^3(t)}{dt^2}$

(b) O-MOMS with $\phi(t) = \beta^3(t) + \frac{1}{42} \frac{d^2\beta^3(t)}{dt^2}$

5.21: Study the FSM part of the cubic B-spline interpolator from Example 5.14, (p. 369) for an $R = 147/160$ rate changer.

(a) Assume that the delays d^k are stored in LUTs or M9Ks tables. What is the required table size if d^k are quantized to

(a1) 8-bit unsigned?

(a2) 16-bit unsigned?

(b) Determine the first five phase values for (a1) and (a2). Is 8 bit sufficient precision?

(c) Assume that the Farrow structure is used, i.e., the delays d^k are computed successively using the Horner scheme. What are the FSM hardware requirements for this solution?

5.22: Use the results from Exercise 5.20 and the fractional rate change design from Example 5.14, (p. 369) to design an $R = 3/4$ rate changer using O-MOMS.

(a) Determine the RAG- n for the FIR compensation filter with the following coefficients: $(-0.0094, 0.0292, -0.0831, 0.2432, -0.7048, 2.0498, -0.7048, 0.2432, \dots) = (-1, 4, -11, 31, -90, 262, -90, 31, -11, 4, -1)/128$. (Hint: add NOF 3)

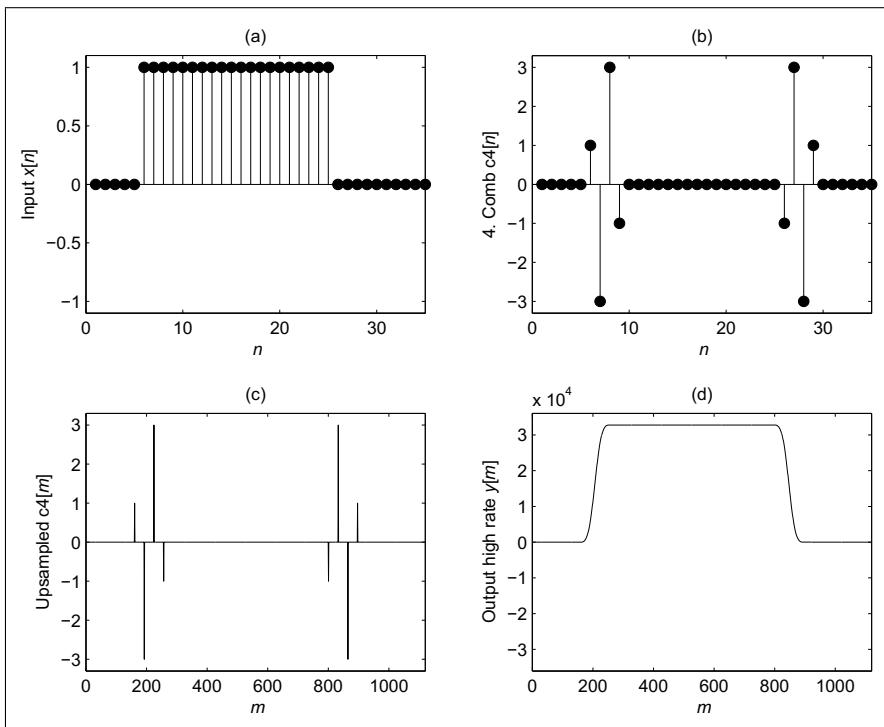


Fig. 5.81. Simulation of the GC4114 CIC interpolator

(b) Replace the IIR filter with the FIR filter from (a) and adjust the Farrow matrix coefficients as determined in Exercise 5.20(b).

(c) Verify the functionality with a triangular test function as in Fig. 5.50, p. 373.

(d) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, embedded multipliers, and M9Ks) for the OMOMS design.

5.23: Use the Farrow matrix (5.60) (p. 366) and the fractional rate change design from Example 5.14, (p. 369) to design an $R = 3/4$ rate changer using B-splines.

(a) Determine the RAG- n for the FIR compensation filter with the following coefficients: $(0.0085, -0.0337, 0.1239, -0.4645, 1.7316, -0.4645, 0.1239, -0.0337, 0.0085) = (1, -4, 16, -59, 222, -59, 16, -4, 1)/128$ (Hint: add NOF 7)

(b) Replace the IIR filter with the FIR filter from (a).

(c) Verify the functionality with a triangular test function as in Fig. 5.50, p. 373.

(d) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, embedded multipliers, and M9Ks) for the B-spline design.

5.24: (a) The GC4114 has a four-stage CIC interpolator with a variable sampling change factor R . Try to download and study the datasheet for the GC4114 from the WWW.

(b) Write a short C or MATLAB program that computes the bit growth $B_k = \log_2(G_k)$ for the CIC interpolator using Hogenauers [127] equation:

$$G_k = \begin{cases} 2^k & k = 1, 2, \dots, S \\ \frac{2^{2S-k}(RD)^{k-S}}{R} & k = S+1, S+2, \dots, 2S \end{cases} \quad (5.114)$$

where D is the delay of the comb, S is number of stages, and R is the interpolation factor. Determine for the GC4114 ($S = 4$ stages, delay comb $D = 1$) the output bit growth for $R=8$, $R=32$, and $R=16\,384$.

- (c) Write a MATLAB program to simulate a four-stage CIC interpolator with delay 1 in the comb and $R=32$ upsampling. Try to match the simulation shown in Fig. 5.81.
- (d) Measure the bit growth for each stage using the program from (c) for a step input and compare the results to (b).

5.25: Using the results from Exercise 5.24

- (a) Design a four-stage CIC interpolator with delay $D = 1$ in the comb and $R = 32$ rate change for 16-bit input. Use for the internal bit width the output bit width you determined in Exercise 5.24(b) for $R=32$. Try to match the MATLAB simulation shown in Fig. 5.81 with the HDL simulation for the input and output.
- (b) Design the four-stage CIC interpolator now with the detailed bit width determined in Exercise 5.24(b). Try to match the MATLAB simulation with the HDL simulation for the input and output.
- (c) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for the two designs from (a) and (b) using:
 - (c1) the device EP2C35F672C6 from the Cyclone II family
 - (c2) the device EP4CE115F29C7 from the Cyclone IV E family

6. Fourier Transforms

The discrete Fourier transform (DFT) and its fast implementation, the fast Fourier transform (FFT), have played a central role in digital signal processing.

DFT and FFT algorithms have been invented (and reinvented) in many variations. As Heideman et al. [174] pointed out, we know that Gauss used an FFT-type algorithm that today we call the Cooley–Tukey FFT. In this chapter we will discuss the most important algorithms summarized in Fig. 6.1.

We will follow the terminology introduced by Burrus [175], who classified FFT algorithms simply by the (multidimensional) index maps of their input and output sequences. We will therefore call all algorithms that do *not* use a multidimensional index map, DFT algorithms, although some of them, such as the Winograd DFT algorithms, enjoy an essentially reduced computational effort. DFT and FFT algorithms do not “stand alone”: the most

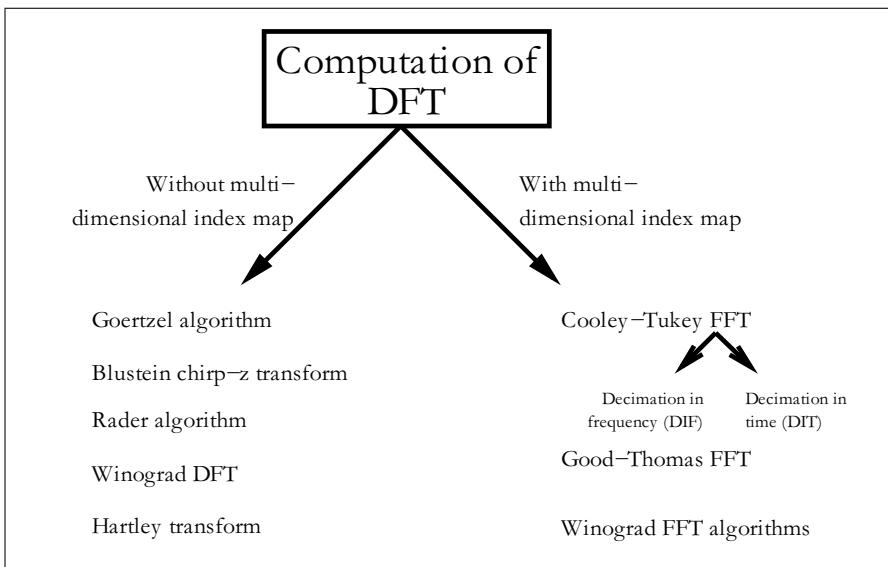


Fig. 6.1. Classifications of DFT and FFT algorithms

efficient implementations often result in a combination of DFT and FFT algorithms. For instance, the combination of the Rader prime algorithm and the Good–Thomas FFT results in excellent VLSI implementations. The literature provides many FFT design examples. We find implementations with PDSPs and ASICs [176–181]. FFTs have also been developed using FPGAs for 1-D [182–184] and 2-D transforms [52, 185].

We will discuss in this chapter the four most important DFT algorithms and the three most often used FFT algorithms, in terms of computational effort, and will compare the different implementation issues. At the end of the chapter, we will discuss Fourier-related transforms, such as the DCT, which is an important tool in image compression (e.g., JPEG, MPEG). We start with a short review of definitions and the most important properties of the DFT.

For more detailed study, students should be aware that DFT algorithms are covered in basic DSP books [5, 88, 186, 187], and a wide variety of FFT books are also available [74, 188–192].

6.1 The Discrete Fourier Transform Algorithms

We will start with a review of the most important DFT properties and will then review basic DFT algorithms introduced by Bluestein, Goertzel, Rader, and Winograd.

6.1.1 Fourier Transform Approximations Using the DFT

The Fourier transform pair is defined by

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft} dt \longleftrightarrow x(t) = \int_{-\infty}^{\infty} X(f)e^{j2\pi ft} df. \quad (6.1)$$

The formulation assumes a continuous signal of infinite duration and bandwidth. For practical representation, we must sample in time and frequency, and amplitudes must be quantized. From an implementation standpoint, we prefer to use a finite number of samples in time and frequency. This leads to the *discrete Fourier transform* (DFT), where N samples are used in time and frequency, according to

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi kn/N} = \sum_{n=0}^{N-1} x[n]W_N^{kn}, \quad (6.2)$$

and the inverse DFT (IDFT) is defined as

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k]e^{j2\pi kn/N} = \frac{1}{N} \sum_{k=0}^{N-1} X[k]W_N^{-kn}, \quad (6.3)$$

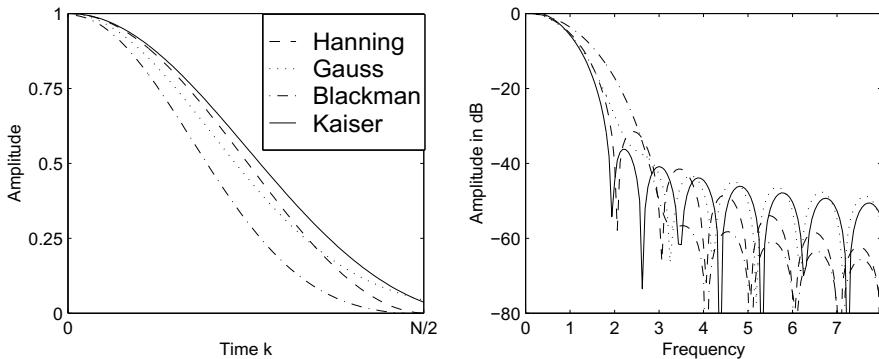


Fig. 6.2. Window functions in time and frequency

or, in vector/matrix notation

$$\mathbf{X} = \mathbf{W}\mathbf{x} \leftrightarrow \mathbf{x} = \frac{1}{N}\mathbf{W}^*\mathbf{X}. \quad (6.4)$$

If we use the DFT to approximate the Fourier spectrum, we must remember the effect of sampling in time and frequency, namely:

- By sampling in *time*, we get a periodic spectrum with the sampling frequency f_s . The approximation of a Fourier transform by a DFT is reasonable only if the frequency components of $x(t)$ are concentrated on a smaller range than the Nyquist frequency $f_s/2$, as stated in the “Shannon sampling theorem.”
- By sampling in the *frequency* domain, the time function becomes periodic, i.e., the DFT assumes the time series to be periodic. If an N -sample DFT is applied to a signal that does not complete an integer number of cycles within an N -sample window, a phenomenon called *leakage* occurs. Therefore, if possible, we should choose the sampling frequency and the analysis window in such a way that it covers an integer number of periods of $x(t)$, if $x(t)$ is periodic.

A more practical alternative for decreasing leakage is the use of a window function that tapers smoothly to zero on both sides. Such window functions were already discussed in the context of FIR filter design in Chap. 3 (see Table 3.2, p. 189). Figure 6.2 shows the time and frequency behavior of some typical windows [128, 193].

An example illustrates the use of a window function.

Example 6.1: Windowing

Figure 6.3a shows a sinusoidal signal that does not complete an integer number of periods in its sample window. The Fourier transform of the signal should ideally include only the two Dirac functions at $\pm\omega_0$, as displayed in Fig. 6.3b. Figures 6.3c and d show the DFT analysis with different windows. We notice that the analysis with the box function has somewhat more ripple

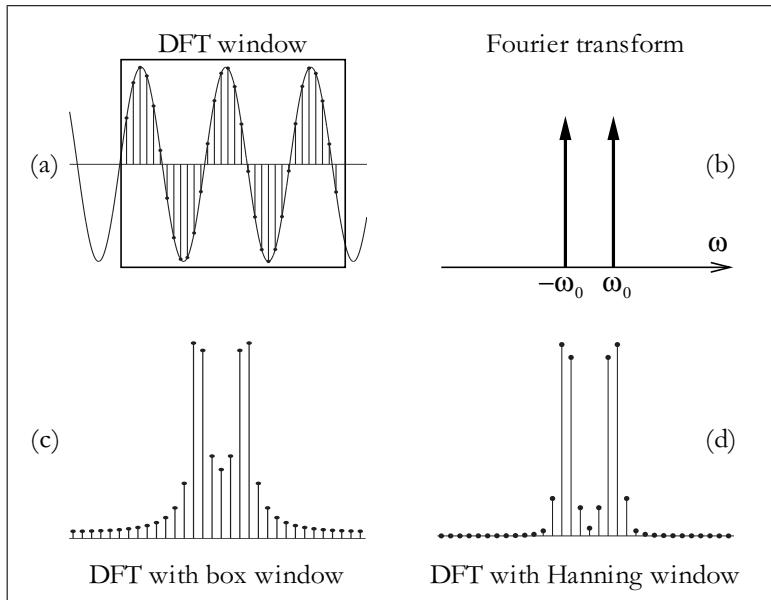


Fig. 6.3. Analysis of periodic function through the DFT, using window functions

than the analysis with the Hanning window. An exact analysis would also show that the main lobe width with Hanning analysis is larger than the width achieved with the box function, i.e., no window.

6.1

6.1.2 Properties of the DFT

The most important properties of the DFT are summarized in Table 6.1. Many properties are identical with the Fourier transform, e.g., the transform is unique (bijective), the superposition applies, and real and imaginary parts are related through the Hilbert transform.

The similarity of the forward and inverse transform leads to an alternative inversion algorithm. Using the vector/matrix notation (6.4) of the DFT

$$\mathbf{X} = \mathbf{W}\mathbf{x} \leftrightarrow \mathbf{x} = \frac{1}{N}\mathbf{W}^*\mathbf{X}, \quad (6.5)$$

we can conclude

$$\mathbf{x}^* = \frac{1}{N}(\mathbf{W}^*\mathbf{X})^* = \frac{1}{N}\mathbf{W}\mathbf{X}^*, \quad (6.6)$$

i.e., we can use the DFT of \mathbf{X}^* scaled by $1/N$ to compute the inverse DFT.

Table 6.1. Theorems of the DFT

Theorem	$x[n]$	$X[k]$
Transform	$x[n]$	$\sum_{n=0}^{N-1} x[n]e^{-j2\pi nk/N}$
Inverse Transform	$\frac{1}{N} \sum_{k=0}^{N-1} X[k]e^{j2\pi nk/N}$	$X[k]$
Superposition	$s_1x_1[n] + s_2x_2[n]$	$s_1X_1[k] + s_2X_2[k]$
Time reversal	$x[-n]$	$X[-k]$
Conjugate complex	$x^*[n]$	$X^*[-k]$
Split		
Real part	$\Re(x[n])$	$(X[k] + X^*[-k])/2$
Imaginary part	$\Im(x[n])$	$(X[k] + X^*[-k])/(2j)$
Real even part	$x_e[n] = (x[n] + x[-n])/2$	$\Re(X[k])$
Real odd part	$x_o[n] = (x[n] - x[-n])/2$	$j\Im(X[k])$
Symmetry	$X[n]$	$Nx[-k]$
Cyclic convolution	$x[n] \circledast f[n]$	$X[k]F[k]$
Multiplication	$x[n] \times f[n]$	$\frac{1}{N} X[k] \circledast F[k]$
Periodic shift	$x[n - d \bmod N]$	$X[k]e^{-j2\pi dk/N}$
Parseval theorem	$\sum_{n=0}^{N-1} x[n] ^2$	$\frac{1}{N} \sum_{k=0}^{N-1} X[k] ^2$

DFT of a Real Sequence

We now turn to some additional computational savings for DFT (and FFT) computations, when the input sequence is real. In this case, we have two options: we can compute with one N -point DFT the DFT of two N -point sequences, or we can compute with an N -point DFT a length $2N$ DFT of a real sequence.

If we use the Hilbert property from Table 6.1, i.e., a real sequence has an even-symmetric real spectrum and an odd imaginary spectrum, the following algorithms can be synthesized [188].

Algorithm 6.2: Length $2N$ Transform with N -point DFT

The algorithm to compute the $2N$ -point DFT $X[k] = X_r[k] + jX_i[k]$ from the time sequence $x[n]$ is as follows:

- 1) Build an N -point sequence $y[n] = x[2n] + jx[2n + 1]$ with $n = 0, 1, \dots, N - 1$.
- 2) Compute $y[n] \circ\bullet Y[k] = Y_r[k] + jY_i[k]$, where $\Re(Y[k]) = Y_r[k]$ is the real and $\Im(Y[k]) = Y_i[k]$ is the imaginary part of $Y[k]$, respectively.
- 3) Compute

$$X_r[k] = \frac{Y_r[k] + Y_r[-k]}{2} + \cos(\pi k/N) \frac{Y_i[k] + Y_i[-k]}{2}$$

$$- \sin(\pi k/N) \frac{Y_r[k] - Y_r[-k]}{2}$$

$$X_i[k] = \frac{Y_i[k] - Y_i[-k]}{2} - \sin(\pi k/N) \frac{Y_i[k] + Y_i[-k]}{2}$$

$$- \cos(\pi k/N) \frac{Y_r[k] - Y_r[-k]}{2}$$

with $k = 0, 1, \dots, N - 1$.

The computational effort, therefore, besides an N -point DFT (or FFT), is 4 N real additions and multiplications, from the twiddle factors $\pm \exp(j\pi k/N)$.

To transform two length- N sequences with a length- N DFT, we use the fact (see Table 6.1) that a real sequence has an even spectrum, while the spectrum of a purely imaginary sequence is odd. This is the basis for the following algorithm.

Algorithm 6.3: Two Length N Transforms with one N -point DFT

The algorithm to compute the N -point DFT $g[n] \circ\bullet G[k]$ and $h[n] \circ\bullet H[k]$ is as follows:

- 1) Build an N -point sequence $y[n] = h[n] + jg[n]$ with $n = 0, 1, \dots, N - 1$.
- 2) Compute $y[n] \circ\bullet Y[k] = Y_r[k] + jY_i[k]$, where $\Re(Y[k]) = Y_r[k]$ is the real and $\Im(Y[k]) = Y_i[k]$ is the imaginary part of $Y[k]$, respectively.
- 3) Compute, finally

$$H[k] = \frac{Y_r[k] + Y_r[-k]}{2} + j \frac{Y_i[k] - Y_i[-k]}{2}$$

$$G[k] = \frac{Y_i[k] + Y_i[-k]}{2} - j \frac{Y_r[k] - Y_r[-k]}{2},$$

with $k = 0, 1, \dots, N - 1$.

The computational effort, therefore, besides an N -point DFT (or FFT), is 2 N real additions, to form the correct two N -point DFTs.

Fast Convolution Using DFT

One of the most frequent applications of the DFT (or FFT) is the computation of convolutions. As with the Fourier transform, the convolution in time is

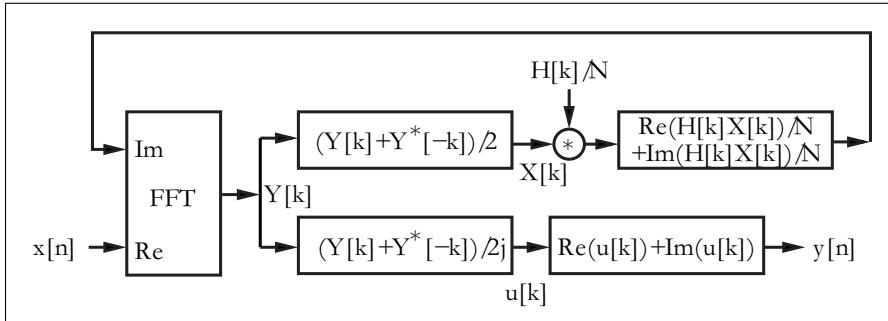


Fig. 6.4. Real convolution using a complex FFT [74]

done by multiplying the two transformed sequences: the two time sequences are transformed in the frequency domain, we compute a (scalar) pointwise product, and we transform the product back into the time domain. The main difference, compared with the Fourier transform, is that now the DFT computes a *cyclic*, and not a linear, convolution. This must be considered when implementing fast convolution with the FFT. This leads to two methods called “overlap save” and “overlap add.” In the overlap save method, we basically discharge the samples at the border that are corrupted by the cyclic convolution. In the overlap add method, we zero-pad the filter and signal in such a way that we can directly add the partial sequences to a common product stream.

Most often the input sequences for the fast convolution are real. An efficient convolution may therefore be accomplished with a real transform, such as the Hartley transform discussed in Exercise 6.15, p. 468. We may also construct an FFT-like algorithm for the Hartley transform, and can get about twice the performance compared with a complex transform [194].

If we wish to utilize an available FFT program, we may use one of the previously discussed Algorithms 6.2 or 6.3, for real sequences. An alternative approach is shown in Fig. 6.4. It shows a similar approach to Algorithm 6.3, where we implemented two N -point transforms with one N -point DFT, but in this case we use the “real” part for a DFT, and the imaginary part for the IDFT, which is needed for the back transformation, according to the convolution theorem.

It is assumed that the DFT of the real-valued filter (i.e., $F[k] = F[-k]^*$) has been computed offline and, in addition, in the frequency domain we need only $N/2$ multiplications to compute $X[k]F[k]$.

6.1.3 The Goertzel Algorithm

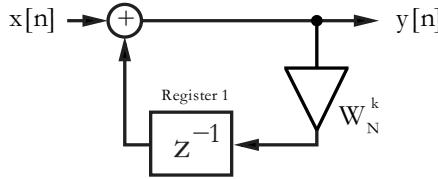
A single spectral component $X[k]$ in the DFT computation is given by

$$X[k] = x[0] + x[1]W_N^k + x[2]W_N^{2k} + \dots + x[N-1]W_N^{(N-1)k}.$$

We can combine all $x[n]$ with the same common factor W_N^k , and get

$$X[k] = x[0] + W_N^k (x[1] + W_N^k (x[2] + \dots + W_N^k x[N-1]) \dots).$$

It can be noted that this results in a possible recursive computation of $X[k]$. This is called the Goertzel algorithm, and is graphically interpreted by Fig. 6.5. The computation of $y[n]$ starts with the last value of the input sequence $x[N-1]$. After step three, a spectrum value of $X[k]$ is available at the output.



Step	$x[n]$	Register 1	$y[n]$
0	$x[3]$	0	$x[3]$
1	$x[2]$	$W_4^k x[3]$	$x[2] + W_4^k x[3]$
2	$x[1]$	$W_4^k x[2] + W_4^{2k} x[3]$	$x[1] + W_4^k x[2] + W_4^{2k} x[3]$
3	$x[0]$	$W_4^k x[1] + W_4^{2k} x[2] + W_4^{3k} x[3]$	$x[0] + W_4^k x[1] + W_4^{2k} x[2] + W_4^{3k} x[3]$

Fig. 6.5. The length-4 Goertzel algorithm

If we have to compute several spectral components, we can reduce the complexity if we combine factors of the type $e^{\pm j2\pi n/N}$. This will result in second order systems having a denominator according to

$$z^2 - 2z \cos\left(\frac{2\pi n}{N}\right) + 1.$$

All complex multiplications are then reduced to real multiplications.

In general, the Goertzel algorithm can be attractive if only a few spectral components have to be computed. For the whole DFT, the effort is of order N^2 , and therefore yields no advantage compared with the direct DFT computation.

6.1.4 The Bluestein Chirp- z Transform

In the Bluestein chirp- z transform (CZT) algorithm, the DFT exponent nk is quadratic expanded to

$$nk = -(k-n)^2/2 + n^2/2 + k^2/2. \quad (6.7)$$

The DFT therefore becomes

$$X[k] = W_N^{k^2/2} \sum_{n=0}^{N-1} (x[n]W_N^{n^2/2}) W_N^{-(k-n)^2/2}. \quad (6.8)$$

This algorithm is graphically interpreted in Fig. 6.6. This results in the following

Algorithm 6.4: Bluestein Chirp- z Algorithm

The computation of the DFT is done in three steps, namely

- 1) N multiplication of $x[n]$ with $W_N^{n^2/2}$
- 2) Linear convolution of $x[n]W_N^{n^2/2} * W_N^{n^2/2}$
- 3) N multiplications with $W_N^{k^2/2}$

For a complete transform, we therefore need a length- N convolution and $2N$ complex multiplications. The advantage, compared with the Rader algorithms, is that there is no restriction to primes in the transform length N . The CZT can be defined for every length.

Narasimha et al. [195] and others have noticed that in the CZT algorithm many coefficients of the FIR filter part are trivial or identical. For instance, the length-8 CZT has an FIR filter of length 14, but there are only four different complex coefficients, as graphically interpreted in Fig. 6.7. These four coefficients are 1, j , and $\pm e^{j22.5^\circ}$, i.e., we only have two nontrivial real coefficients to implement.

It may be of general interest what the *maximum* DFT length for a fixed number C_N of (complex) coefficients is. This is shown in the following table.

DFT length	8	12	16	24	40	48	72	80	120	144	168	180	240	360	504
C_N	4	6	7	8	12	14	16	21	24	28	32	36	42	48	64

As mentioned before, the number of different complex coefficients does not directly correspond to the implementation effort, because some coefficients

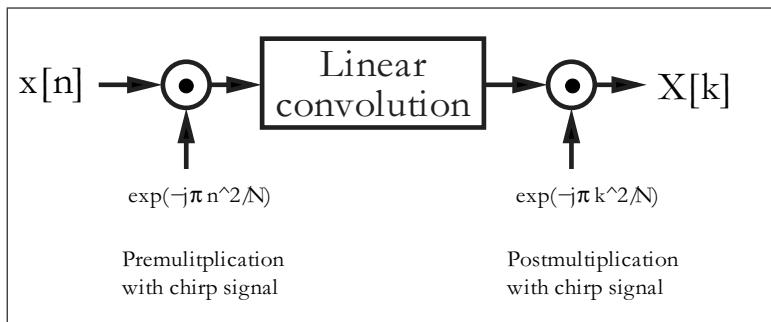


Fig. 6.6. The Bluestein chirp- z algorithm

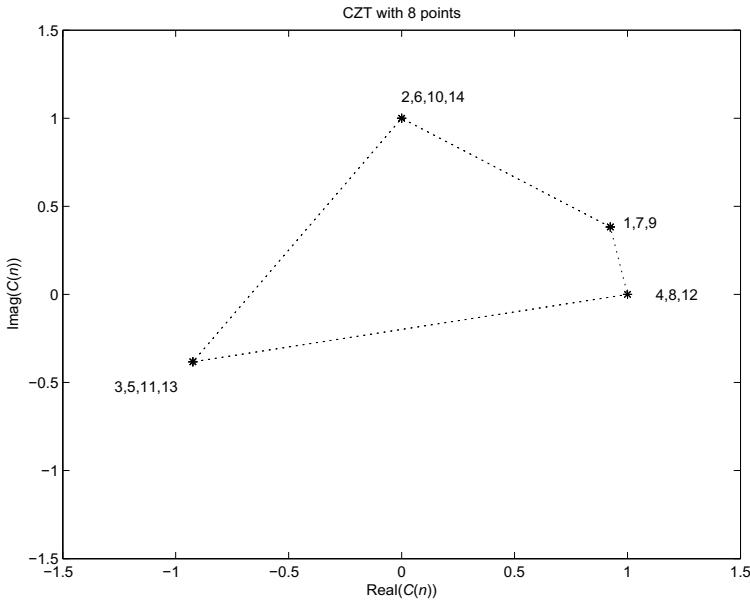


Fig. 6.7. CZT coefficients $C(n) = e^{j2\pi \frac{n^2/2 \bmod 8}{8}}$; $n = 1, 2, \dots, 14$

may be trivial (i.e., ± 1 or $\pm j$) or may show symmetry. In particular, the power-of-two length transform enjoys many symmetries, as can be seen from Fig. 6.8. If we compute the maximum DFT length for a specific number of nontrivial real coefficients, we find as maximum-length transforms:

DFT length	10	16	20	32	40	48	50	80	96	160	192
sin/cos	2	3	5	6	8	9	10	11	14	20	25

Length 16 and 32 are therefore the maximum length DFTs with only three and six real multipliers, respectively.

In general, power-of-two lengths are popular FFT building blocks, and the following table therefore shows, for length $N = 2^n$, the effort when implementing the CZT filter in transposed form.

N	C_N	sin/cos	CSD adder	RAG adder	NOFs for 14-bit coefficients
8	4	2	23	7	3,5,33,49,59
16	7	3	91	8	3,25,59,63,387
32	12	6	183	13	3,25,49,73,121,375
64	23	11	431	18	5,25,27,93,181,251,7393
128	44	22	879	31	5,15,25,175,199,319,403,499,1567
256	87	42	1911	49	5,25,765,1443,1737,2837,4637

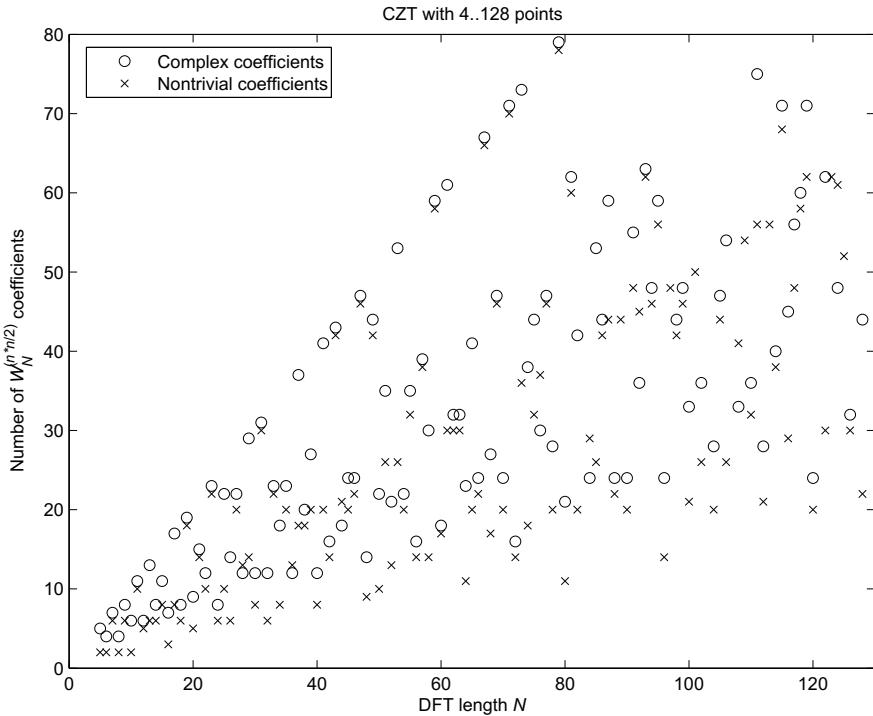


Fig. 6.8. Number of complex coefficients and nontrivial real multiplications for the CZT

The first column shows the DFT length N . The second column shows the total number of complex exponentials C_N . The worst-case effort for C_N complex coefficients is that $2C_N$ real, nontrivial coefficients must be implemented. The actual number of different nontrivial real coefficients is shown in column three. We note when comparing columns two and three that for power-of-two lengths the symmetry and trivial coefficients reduce the number of nontrivial coefficients. The next columns show, for CZT DFTs up to length 256, the effort (i.e., number of adders) for a 15-bit (14-bit unsigned plus sign bit) coefficient precision implementation, using the CSD and RAG algorithms (discussed in Chap. 2), respectively. For CSD coding no coefficient symmetry is considered and the adder count is quite high. We note that the RAG algorithm when compared with CSD can essentially reduce the effort for DFT lengths larger than 16.

6.1.5 The Rader Algorithm

The Rader algorithm [196, 197] to compute a DFT,

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk} \quad k, n \in \mathbb{Z}_N; \quad \text{ord}(W_N) = N \quad (6.9)$$

is defined only for prime length N . We first compute the DC component with

$$X[0] = \sum_{n=0}^{N-1} x[n]. \quad (6.10)$$

Because $N = p$ is a prime, we know from the discussion in Chap. 2 (p. 71) that there is a primitive element, a *generator* g , that generates all elements of n and k in the field \mathbb{Z}_p , excluding zero, i.e., $g^k \in \mathbb{Z}_p/\{0\}$. We substitute n by $g^n \bmod N$ and k with $g^k \bmod N$, and get the following index transform:

$$X[g^k \bmod N] - x[0] = \sum_{n=0}^{N-2} x[g^n \bmod N] W_N^{g^{n+k} \bmod N} \quad (6.11)$$

for $k \in \{1, 2, 3, \dots, N-1\}$. We note that the right side of (6.11) is a cyclic convolution, i.e.,

$$\begin{aligned} & [x[g^0 \bmod N], x[g^1 \bmod N], \dots, x[g^{N-2} \bmod N]] \\ & \quad \circledast \left[W_N, W_N^g, \dots, W_N^{g^{N-2} \bmod N} \right]. \end{aligned} \quad (6.12)$$

An example with $N = 7$ demonstrates the Rader algorithms.

Example 6.5: Rader Algorithms for $N = 7$

For $N = 7$, we know that $g = 3$ is a primitive element (see, for instance, [5], Table B.7), and the index transform is

$$[3^0, 3^1, 3^2, 3^3, 3^4, 3^5] \bmod 7 \equiv [1, 3, 2, 6, 4, 5]. \quad (6.13)$$

We first compute the DC component

$$X[0] = \sum_{n=0}^6 x[n] = x[0] + x[1] + x[2] + x[3] + x[4] + x[5] + x[6],$$

and in the second step, the cyclic convolution of $X[k] - x[0]$

$$[x[1], x[3], x[2], x[6], x[4], x[5]] \circledast [W_7, W_7^3, W_7^2, W_7^6, W_7^4, W_7^5],$$

or in matrix notation

$$\begin{bmatrix} X[1] \\ X[3] \\ X[2] \\ X[6] \\ X[4] \\ X[5] \end{bmatrix} = \begin{bmatrix} W_7^1 & W_7^3 & W_7^2 & W_7^6 & W_7^4 & W_7^5 \\ W_7^3 & W_7^2 & W_7^6 & W_7^4 & W_7^5 & W_7^1 \\ W_7^2 & W_7^6 & W_7^4 & W_7^5 & W_7^1 & W_7^3 \\ W_7^6 & W_7^4 & W_7^5 & W_7^1 & W_7^3 & W_7^2 \\ W_7^4 & W_7^5 & W_7^1 & W_7^3 & W_7^2 & W_7^6 \\ W_7^5 & W_7^1 & W_7^3 & W_7^2 & W_7^6 & W_7^4 \end{bmatrix} \begin{bmatrix} x[1] \\ x[3] \\ x[2] \\ x[6] \\ x[4] \\ x[5] \end{bmatrix} + \begin{bmatrix} x[0] \\ x[0] \\ x[0] \\ x[0] \\ x[0] \\ x[0] \end{bmatrix}. \quad (6.14)$$

This is graphically interpreted using an FIR filter in Fig. 6.9.

We now verify the $p = 7$ Rader DFT formula, using a test triangular signal $x[n] = 10\lambda[n]$ (i.e., a triangle with step size 10). Directly interpreting (6.14), one obtains

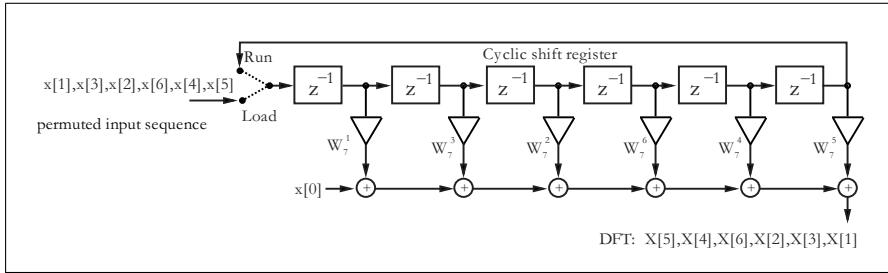


Fig. 6.9. Length $p = 7$ Rader prime-factor DFT implementation

$$\begin{bmatrix} X[1] \\ X[3] \\ X[2] \\ X[6] \\ X[4] \\ X[5] \end{bmatrix} = \begin{bmatrix} W_7^1 & W_7^3 & W_7^2 & W_7^6 & W_7^4 & W_7^5 \\ W_7^3 & W_7^2 & W_7^6 & W_7^4 & W_7^5 & W_7^1 \\ W_7^2 & W_7^6 & W_7^4 & W_7^5 & W_7^1 & W_7^3 \\ W_7^6 & W_7^4 & W_7^5 & W_7^1 & W_7^3 & W_7^2 \\ W_7^4 & W_7^5 & W_7^1 & W_7^3 & W_7^2 & W_7^6 \\ W_7^5 & W_7^1 & W_7^3 & W_7^2 & W_7^6 & W_7^4 \end{bmatrix} \begin{bmatrix} 20 \\ 40 \\ 30 \\ 70 \\ 50 \\ 60 \end{bmatrix} + \begin{bmatrix} 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \end{bmatrix}$$

$$= \begin{bmatrix} -35 + j72 \\ -35 + j8 \\ -35 + j28 \\ -35 - j72 \\ -35 - j8 \\ -35 - 28 \end{bmatrix}.$$

The value of $X[0]$ is the sum of the time series, which is $10+20+\cdots+70 = 280$.

6.5

In addition, in the Rader algorithms we may use the symmetries of the complex pairs $e^{\pm j2k\pi/N}$, $k \in [0, N/2]$, to build more-efficient FIR realizations (Exercise 6.5, p. 466). Implementing a Rader prime-factor DFT is equivalent to implementing an FIR filter, which we discussed in Chap. 3. In order to implement a fast FIR filter, a fully pipelined DA or the transposed filter structure using the RAG algorithm is attractive. The RAG FPGA implementation is illustrated in the following example.

Example 6.6: Rader FPGA Implementation

An RAG implementation of the length-7 Rader algorithm is accomplished as follows. The first step is quantizing the coefficients. Assuming that the input values and coefficients are to be represented as a signed 8-bit word, the quantized coefficients are:

k	0	1	2	3	4	5	6
$\text{Re}\{256 \times W_7^k\}$	256	160	-57	-231	-231	-57	160
$\text{Im}\{256 \times W_7^k\}$	0	-200	-250	-111	111	250	200

A direct-form implementation of all the individual coefficients would (consulting Table 2.3, p. 68) consume 24 adders for the constant coefficient multipliers. Using the transposed structure, the individual coefficient implemen-

tation effort is reduced to 11 adders by exploiting the fact that several coefficients differ only in sign. Optimizing further (reduced adder graph, see Fig. 2.4, p. 67), the number of adders reaches a minimum of seven (see Factor: PROCESS and Coeffs: PROCESS below). This is more than a three times improvement over the direct FIR architecture. The following VHDL code¹ illustrates a possible implementation of the length-7 Rader DFT, using transposed FIR filters:

```

PACKAGE n_bit_int IS      -----> User defined types
    SUBTYPE U4 IS INTEGER RANGE 0 TO 15;
    SUBTYPE S8 IS INTEGER RANGE -2**7 TO 2**7-1;
    SUBTYPE S11 IS INTEGER RANGE -2**10 TO 2**10-1;
    SUBTYPE S19 IS INTEGER RANGE -2**18 TO 2**18-1;
    TYPE A0_5S19 IS ARRAY (0 to 5) OF S19;
END n_bit_int;

LIBRARY work;
USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
-----
ENTITY rader7 IS          -----> Interface
    PORT ( clk      : IN STD_LOGIC;      -- System clock
           reset    : IN STD_LOGIC;      -- Asynchronous reset
           x_in     : IN S8;          -- Real system input
           y_real   : OUT S11;        -- Real system output
           y_imag   : OUT S11);       -- Imaginary system output
END rader7;
-----
ARCHITECTURE fpga OF rader7 IS

    SIGNAL count      : U4;           -- Clock cycle counter
    TYPE STATE_TYPE IS (Start, Load, Run);
    SIGNAL state      : STATE_TYPE ; -- State variable
    SIGNAL accu       : S11 := 0;     -- Signal for X[0]
    SIGNAL real, imag : A0_5S19;     -- Tapped delay line array
    SIGNAL x57, x111, x160, x200, x231, x250 : S19 := 0;
                                         -- The (unsigned) filter coefficients
    SIGNAL x5, x25, x110, x125, x256 : S19 := 0;
                                         -- Auxiliary filter coefficients
    SIGNAL x, x_0 : S8 := 0;         -- Signals for x[0]

BEGIN

    States: PROCESS (reset, clk)----- FSM for RADER filter
    BEGIN
        IF reset = '1' THEN           -- Asynchronous reset
            state <= Start; accu <= 0;

```

¹ The equivalent Verilog code `rader7.v` for this example can be found in Appendix A on page 851. Synthesis results are shown in Appendix B on page 881.

```

count <= 0; y_real <= 0; y_imag <= 0;
ELSIF rising_edge(clk) THEN
  CASE state IS
    WHEN Start =>           -- Initialization step
      state <= Load;
      count <= 1;
      x_0 <= x_in;          -- Save x[0]
      accu <= 0;            -- Reset accumulator for X[0]
      y_real <= 0;
      y_imag <= 0;
    WHEN Load => -- Apply x[5],x[4],x[6],x[2],x[3],x[1]
      IF count = 8 THEN     -- Load phase done ?
        state <= Run;
      ELSE
        state <= Load;
        accu <= accu + x ;
      END IF;
      count <= count + 1;
    WHEN Run => -- Apply again x[5],x[4],x[6],x[2],x[3]
      IF count = 15 THEN   -- Run phase done ?
        y_real <= accu;    -- X[0]
        y_imag <= 0;       -- Only re inputs i.e. Im(X[0])=0
        state <= Start;    -- Output of result
        count <= 0;
      ELSE                  -- and start again
        y_real <= real(0) / 256 + x_0;
        y_imag <= imag(0) / 256;
        state <= Run;
        count <= count + 1;
      END IF;
    END CASE;
  END IF;
END PROCESS States;

-- Structure of the two FIR filters in transposed form
Structure: PROCESS(clk, reset, x_in, real, imag,
                   x57, x111, x160, x200, x231, x250)
BEGIN
  IF reset = '1' THEN           -- Asynchronous clear
    FOR K IN 0 TO 5 LOOP
      real(k) <= 0; imag(K) <= 0;
    END LOOP;
    x <= 0;
  ELSIF rising_edge(clk) THEN
    x <= x_in;
    -- Real part of FIR filter in transposed form
    real(0) <= real(1) + x160 ; -- W^1
    real(1) <= real(2) - x231 ; -- W^3
    real(2) <= real(3) - x57  ; -- W^2
    real(3) <= real(4) + x160 ; -- W^6
    real(4) <= real(5) - x231 ; -- W^4
    real(5) <= -x57  ;         -- W^5
  END IF;
END Structure;

```

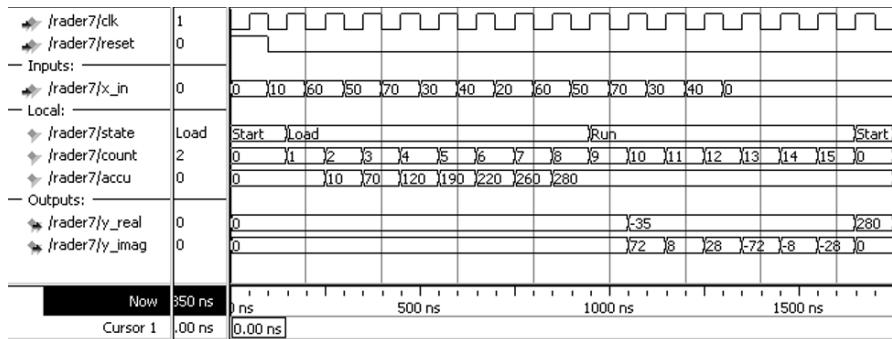


Fig. 6.10. VHDL simulation of a seven-point Rader algorithm

```

-- Imaginary part of FIR filter in transposed form
imag(0) <= imag(1) - x200 ; -- W^1
imag(1) <= imag(2) - x111 ; -- W^3
imag(2) <= imag(3) - x250 ; -- W^2
imag(3) <= imag(4) + x200 ; -- W^6
imag(4) <= imag(5) + x111 ; -- W^4
imag(5) <= x250; -- W^5

END IF;
END PROCESS Structure;

Coeffs: PROCESS(clk, reset, x, x5, x25, x125, x110, x256)
BEGIN
    -- Note that all signals are globally defined
    IF reset = '1' THEN
        -- Asynchronous clear
        x160 <= 0; x200 <= 0; x250 <= 0;
        x57 <= 0; x111 <= 0; x231 <= 0;
    ELSIF rising_edge(clk) THEN
        -- Compute the filter coefficients and use FFs
        x160 <= x5 * 32;
        x200 <= x25 * 8;
        x250 <= x125 * 2;
        x57 <= x25 + x * 32;
        x111 <= x110 + x;
        x231 <= x256 - x25;
    END IF;
END PROCESS Coeffs;

Factors: PROCESS (x, x5, x25)      -- Note that all signals
BEGIN
    -- are globally defined
    -- Compute the auxiliary factor for RAG without an FF
    x5 <= x * 4 + x;
    x25 <= x5 * 4 + x5;
    x110 <= x25 * 4 + x5 * 2;
    x125 <= x25 * 4 + x25;
    x256 <= x * 256;
END PROCESS Factors;

END fpga;

```

The design consists of four blocks of statements within the four **PROCESS** statements. The first – **Stages: PROCESS** – is the state machine, which distinguishes the three processing phases, **Start**, **Load**, and **Run**. The second – **Structure: PROCESS** – defines the two FIR filter paths, real and imaginary. The third item implements the multiplier block using the reduced adder graph. The forth block – **Factor: PROCESS** – implements the unregistered factors of the RAG algorithm. It can be seen that all coefficients are realized by using six adders and one subtractor. The design uses 428 LEs, no embedded multiplier, and has an **Fmax**=138.45 MHz registered performance using the TimeQuest slow 85C model. Figure 6.10 displays simulation results using Quartus II for a triangular input sequence $x[n] = \{10, 20, 30, 40, 50, 60, 70\}$. Note that the input and output sequences, starting at 1 μ s, occur in permuted order, and negative results appear signed if we use the signed data type in the simulator. Finally, at 1.7 μ s, $X[0] = 280$ is forwarded to the output and **rader7** is ready to process the next input frame.

6.6

Because the Rader algorithm is restricted to prime lengths there is less symmetry in the coefficients, compared with the CZT. The following table shows, for primes length $2^n \pm 1$, the implementation effort of the circular filter in transposed form:

DFT length	sin/cos	CSD adder	RAG adder	NOFs for 14-bit coefficients
7	6	52	13	7,11,31,59,101,177,319
17	16	138	23	3,35,103,415,1153,1249,8051
31	30	244	38	3,9,133,797,877,975,1179,3235
61	60	496	66	5,39,51,205,265,3211
127	124	1060	126	5

The first column shows the cyclic convolution length N , which is also the number of complex coefficients. Comparing column two and the worst case with $2N$ real sin/cos coefficients, we see that symmetry and trivial coefficients reduce the number of nontrivial coefficients by a factor of 2. The next two columns show the effort for a 14-bit (plus sign) coefficient precision implementation using CSD or RAG algorithms, respectively. The last column shows the auxiliary coefficient, i.e., NOFs used by RAG. Note the advantage of RAG for longer filters. It can be seen from this table that the effort for CSD-type filters can be estimated by $BN/2$, where B is the coefficient bit width (14 in this table) and N is the filter length. For RAG, the effort (i.e., the number of adders) is only N , i.e., a factor $B/2$ improvement over CSD for longer filters (for $B = 14$, a factor $\approx 14/2 = 7$ of improvement). For longer filters, RAG needs only one additional adder for each additional coefficient, because the already-synthesized coefficient produces a dense grid of small coefficients.

6.1.6 The Winograd DFT Algorithm

The first algorithm with a reduced number of multiplications necessary we want to discuss is the *Winograd DFT* algorithm. The Winograd algorithm is a combination of the Rader algorithm (which translates the DFT into a cyclic convolution), and Winograd's [124] short convolution algorithm, which we have already used to implement fast-running FIR filters (see Sect. 5.2.2, p. 315).

The length is therefore restricted to primes or powers of primes. Table 6.2 gives an overview of the number of arithmetic operations necessary.

Table 6.2. Effort for the Winograd DFT with real inputs. Trivial multiplications are those by ± 1 or $\pm j$. For complex inputs, the number of operations is twice as large

Block length	Total number of real multiplications	Total number nontrivial multiplications	Total number of real additions
2	2	0	2
3	3	2	6
4	4	0	8
5	6	5	17
7	9	8	36
8	8	2	26
9	11	10	44
11	21	20	84
13	21	20	94
16	18	10	74
17	36	35	157
19	39	38	186

The following example for $N = 5$ demonstrates the steps to build a Winograd DFT algorithm.

Example 6.7: $N = 5$ Winograd DFT Algorithm

An alternative representation of the Rader algorithm, using $X[0]$ instead of $x[0]$, is given by [5]

$$\begin{aligned} X[0] &= \sum_{n=0}^4 x[n] = x[0] + x[1] + x[2] + x[3] + x[4] \\ X[k] - X[0] &= [x[1], x[2], x[4], x[3]] \circledast [W_5 - 1, W_5^2 - 1, W_5^4 - 1, W_5^3 - 1] \\ k &= 1, 2, 3, 4. \end{aligned}$$

If we implement the cyclic convolution of length 4 with a Winograd algorithm that costs only five nontrivial multiplications, we get the following algorithm:

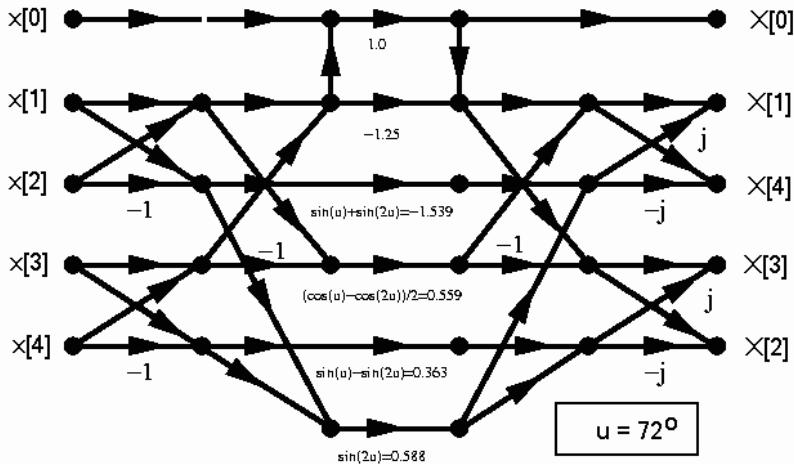


Fig. 6.11. Winograd 5-point DFT signal flow graph

$$\begin{aligned}
 X[k] &= \sum_{n=0}^4 x[n] e^{-j2\pi kn/5} \quad k = 0, 1, \dots, 4 \\
 \begin{bmatrix} X[0] \\ X[4] \\ X[3] \\ X[2] \\ X[1] \end{bmatrix} &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & -1 & -1 \\ 1 & 1 & 1 & -1 & 0 \end{bmatrix} \\
 &\times \text{diag}\left(1, \frac{1}{2}(\cos(2\pi/5) + \cos(4\pi/5)) - 1, \right. \\
 &\quad \left. \frac{1}{2}(\cos(2\pi/5) - \cos(4\pi/5)), j \sin(2\pi/5), \right. \\
 &\quad \left. j(-\sin(2\pi/5) + \sin(4\pi/5)), j(\sin(2\pi/5) + \sin(4\pi/5)) \right) \\
 &\times \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & -1 & -1 & 1 \\ 0 & 1 & -1 & 1 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & -1 & 1 & 0 \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ x[2] \\ x[3] \\ x[4] \end{bmatrix}.
 \end{aligned}$$

The total computational effort is therefore only 5 or 10 real nontrivial multiplications for real or imaginary input sequences $x[n]$, respectively. The signal flow graph shown in Fig. 6.11 shows also how to implement the additions in an efficient fashion.

6.7

It is quite convenient to use a matrix notation for the Winograd DFT algorithm, and so we get

$$\mathbf{W}_{N_l} = \mathbf{C}_l \times \mathbf{B}_l \times \mathbf{A}_l, \quad (6.15)$$

where \mathbf{A}_l incorporates the input addition, \mathbf{B}_l is the diagonal matrix with the Fourier coefficients, and \mathbf{C}_l includes the output additions. The only disadvantage is that now it is not as easy to define the exact steps of the short convolution algorithms, because the sequence in which input and output additions are computed is lost with this matrix representation.

This combination of Rader algorithms and a short Winograd convolution, known as the Winograd DFT algorithm, will be used later, together with index mapping to introduce the Winograd FFT algorithm. This is the FFT algorithm with the least number of real multiplications among all known FFT algorithms.

6.2 The Fast Fourier Transform (FFT) Algorithms

As mentioned in the introduction of this chapter, we use the terminology introduced by Burrus [175], who classified all FFT algorithms simply by different (multidimensional) index maps of the input and output sequences. These are based on a transform of the length N DFT (6.2)

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk} \quad (6.16)$$

into a multidimensional $N = \prod_l N_l$ representation. It is, in general, sufficient to discuss only the two-factor case, because higher dimensions can be built simply by iteratively replacing again one of these factors. To simplify our representation we will therefore discuss the three FFT algorithms presented only in terms of a two-dimensional index transform.

We transform the (time) index n with

$$n = An_1 + Bn_2 \bmod N \quad \begin{cases} 0 \leq n_1 \leq N_1 - 1 \\ 0 \leq n_2 \leq N_2 - 1, \end{cases} \quad (6.17)$$

where $N = N_1 N_2$, and $A, B \in \mathbb{Z}$ are constants that must be defined later. Using this index transform, a two-dimensional mapping $f : \mathbb{C}^N \rightarrow \mathbb{C}^{N_1 \times N_2}$ of the data is built, according to

$$\begin{aligned} & [x[0] \ x[1] \ x[2] \cdots x[N-1]] \\ &= \begin{bmatrix} x[0,0] & x[0,1] & \cdots & x[0,N_2-1] \\ x[1,0] & x[1,1] & \cdots & x[1,N_2-1] \\ \vdots & \vdots & \ddots & \vdots \\ x[N_1-1,0] & x[N_1-1,1] & \cdots & x[N_1-1,N_2-1] \end{bmatrix}. \end{aligned} \quad (6.18)$$

Applying another index mapping k to the output (frequency) domain yields

$$k = Ck_1 + Dk_2 \bmod N \quad \begin{cases} 0 \leq k_1 \leq N_1 - 1 \\ 0 \leq k_2 \leq N_2 - 1, \end{cases} \quad (6.19)$$

where $C, D \in \mathbb{Z}$ are constants that must be defined later. Because the DFT is bijective, we must choose A, B, C , and D in such a way that the transform representation is still unique, i.e., a bijective projection. Burrus [175] has determined the general conditions for how to choose A, B, C , and D for specific N_1 and N_2 such that the mapping is bijective (see Exercises 6.7 and 6.8, p. 466). The transforms given in this chapter are all unique.

An important point in distinguishing different FFT algorithms is the question of whether N_1 and N_2 are allowed to have a common factor, i.e., $\gcd(N_1, N_2) > 1$, or whether the factors must be coprime. Sometimes algorithms with $\gcd(N_1, N_2) > 1$ are referred to as *common-factor algorithms* (CFAs), and algorithms with $\gcd(N_1, N_2) = 1$ are called *prime-factor algorithms* (PFAs). A CFA algorithm discussed in the following is the Cooley–Tukey FFT, while the Good–Thomas and Winograd FFTs are of the PFA type. It should be emphasized that the Cooley–Tukey algorithm may indeed realize FFTs with two factors, $N = N_1 N_2$, which are coprime, and that for a PFA the factors N_1 and N_2 must only be coprime, i.e., they must *not* be primes themselves. A transform of length $N = 12$ factored with $N_1 = 4$ and $N_2 = 3$, for instance, can therefore be used for both CFA FFTs and PFA FFTs!

6.2.1 The Cooley–Tukey FFT Algorithm

The Cooley–Tukey FFT is the most universal of all FFT algorithms, because any factorization of N is possible. The most popular Cooley–Tukey FFTs are those where the transform length N is a power of a basis r , i.e., $N = r^\nu$. These algorithms are often referred to as radix- r algorithms.

The index transform suggested by Cooley and Tukey (and earlier by Gauss) is also the simplest index mapping. Using (6.17) we have $A = N_2$ and $B = 1$, and the following mapping results

$$n = N_2 n_1 + n_2 \quad \begin{cases} 0 \leq n_1 \leq N_1 - 1 \\ 0 \leq n_2 \leq N_2 - 1. \end{cases} \quad (6.20)$$

From the valid range of n_1 and n_2 , we conclude that the modulo reduction given by (6.17) need not be explicitly computed.

For the inverse mapping from (6.19) Cooley and Tukey, choose $C = 1$ and $D = N_1$, and the following mapping results:

$$k = k_1 + N_1 k_2 \quad \begin{cases} 0 \leq k_1 \leq N_1 - 1 \\ 0 \leq k_2 \leq N_2 - 1. \end{cases} \quad (6.21)$$

The modulo computation can also be skipped in this case. If we now substitute n and k in W_N^{nk} according to (6.20) and (6.21), respectively, we find

$$W_N^{nk} = W_N^{N_2 n_1 k_1 + N_1 N_2 n_1 k_2 + n_2 k_1 + N_1 n_2 k_2}. \quad (6.22)$$

Because W is of order $N = N_1 N_2$, it follows that $W_N^{N_1} = W_{N_2}$ and $W_N^{N_2} = W_{N_1}$. This simplifies (6.22) to

$$W_N^{nk} = W_{N_1}^{n_1 k_1} W_N^{n_2 k_1} W_{N_2}^{n_2 k_2}. \quad (6.23)$$

If we now substitute (6.23) in the DFT from (6.16) it follows that:

$$X[k_1, k_2] = \sum_{n_2=0}^{N_2-1} W_{N_2}^{n_2 k_2} \underbrace{\left(W_N^{n_2 k_1} \sum_{n_1=0}^{N_1-1} x[n_1, n_2] W_{N_1}^{n_1 k_1} \right)}_{\text{N}_1\text{-point transform}} \quad (6.24)$$

$$= \underbrace{\sum_{n_2=0}^{N_2-1} W_{N_2}^{n_2 k_2} \bar{x}[n_2, k_1]}_{\text{N}_2\text{-point transform}}. \quad (6.25)$$

We now can define the complete Cooley–Tukey algorithm

Algorithm 6.8: Cooley–Tukey Algorithm

An $N = N_1 N_2$ -point DFT can be done using the following steps:

- 1) Compute an index transform of the input sequence according to (6.20).
- 2) Compute the N_2 DFTs of length N_1 .
- 3) Apply the twiddle factors $W_N^{n_2 k_1}$ to the output of the first transform stage.
- 4) Compute N_1 DFTs of length N_2 .
- 5) Compute an index transform of the output sequence according to (6.21).

The following length-12 transform demonstrates these steps.

Example 6.9: Cooley–Tukey FFT for $N = 12$

Assume $N_1 = 4$ and $N_2 = 3$. It follows then that $n = 3n_1 + n_2$ and $k = k_1 + 4k_2$, and we can compute the following tables for the index mappings:

n_2	n_1				k_2	k_1			
	0	1	2	3		0	1	2	3
0	$x[0]$	$x[3]$	$x[6]$	$x[9]$	0	$X[0]$	$X[1]$	$X[2]$	$X[3]$
1	$x[1]$	$x[4]$	$x[7]$	$x[10]$	1	$X[4]$	$X[5]$	$X[6]$	$X[7]$
2	$x[2]$	$x[5]$	$x[8]$	$x[11]$	2	$X[8]$	$X[9]$	$X[10]$	$X[11]$

With the help of this transform we can construct the signal flow graph shown in Fig. 6.12. It can be seen that first we must compute three DFTs with four points each, followed by the multiplication with the twiddle factors, and finally we compute four DFTs each having length 3.

6.9

For direct computation of the 12-point DFT, a total of $12^2 = 144$ complex multiplications and $11^2 = 121$ complex additions are needed. To compute the Cooley–Tukey FFT with the same length we need a total of 12 complex

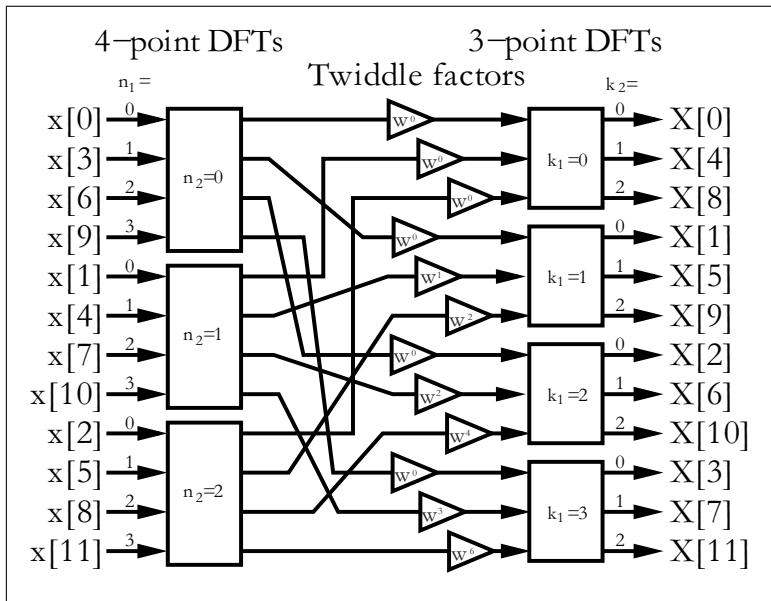


Fig. 6.12. Cooley–Tukey FFT for $N = 12$

multiplication for the twiddle factors, of which 8 are trivial (i.e., ± 1 or $\pm j$) multiplications. According to Table 6.2 (p. 434), the length-4 DFTs can be computed using 8 real additions and no multiplications. For the length-3 DFTs, we need 4 multiplications and 6 additions. If we implement the (fixed coefficient) complex multiplications using 3 additions and 3 multiplications (see Algorithm 6.10, p. 442), and consider that $W^0 = 1, W^3 = -j$ and $W^6 = -1$ are trivial, the total effort for the 12-point Cooley–Tukey FFT is given by

$$3 \times 16 + 4 \times 3 + 4 \times 12 = 108 \text{ real additions and}$$

$$4 \times 3 + 4 \times 4 = 28 \text{ real multiplications.}$$

For the direct implementation we would need $2 \times 11^2 + 12^2 \times 3 = 674$ real additions and $12^2 \times 3 = 432$ real multiplications. It is now obvious why the Cooley–Tukey algorithm is called the “fast Fourier transform” (FFT).

Radix- r Cooley–Tukey Algorithm

One important fact that distinguishes the Cooley–Tukey algorithm from other FFT algorithms is that the factors for N can be chosen arbitrarily. It is therefore possible to use a radix- r algorithm in which $N = r^S$. The most popular algorithms are those of basis $r = 2$ or $r = 4$, because the necessary basic DFTs can, according to Table 6.2 (p. 434), be implemented without

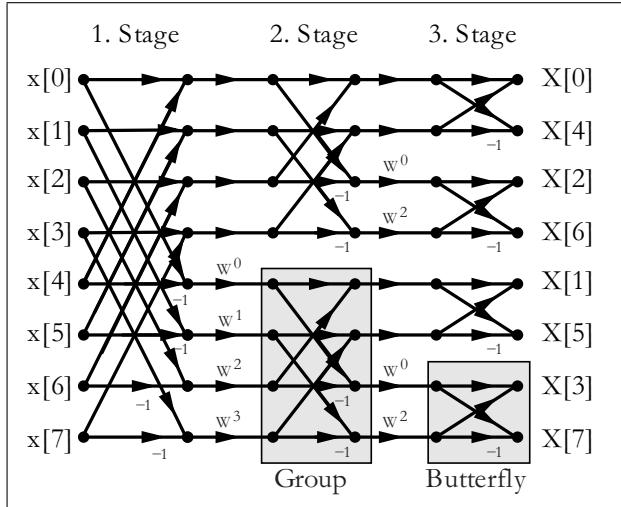


Fig. 6.13. Decimation-in-frequency algorithm of length-8 for radix-2

any multiplications. For $r = 2$ and S stages, for instance, the following index mapping results:

$$n = 2^{S-1}n_1 + \cdots + 2n_{S-1} + n_S \quad (6.26)$$

$$k = k_1 + 2k_2 + \cdots + 2^{S-1}k_S. \quad (6.27)$$

For $S > 2$ a common practice is that in the signal flow graph a 2-point DFT is represented with a *Butterfly*, as shown in Fig. 6.13 for an 8-point transform. The signal flow graph representation has been simplified by using the fact that all arriving arrows at a node are added, while the constant coefficient multiplications are symbolized through a factor at an arrow. A radix- r algorithm has $\log_r(N)$ *stages*, and for each *group* the same type of twiddle factor occurs.

It can be seen from the signal flow graph in Fig. 6.13 that the computation can be done “*in-place*,” i.e., the memory location used by a butterfly can be overwritten, because the data are no longer needed in the next computational steps. The total number of twiddle factor multiplications for the radix-2 transform is given by

$$\log_2(N)N/2, \quad (6.28)$$

because only every second arrow has a twiddle factor.

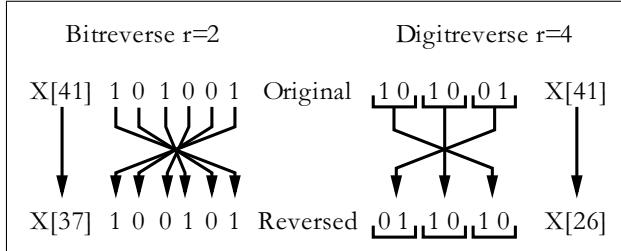
Because the algorithm shown in Fig. 6.13 starts in the frequency domain to split the original DFT into shorter DFTs, this algorithm is called a decimation-in-frequency (DIF) algorithm. The input values typically occur in natural order, while the index of the frequency values is in bit-reversed order. Table 6.3 shows the characteristic values of a DIF radix-2 algorithm.

Table 6.3. Radix-2 FFT with frequency decimation

	Stage 1	Stage 2	Stage 3	...	Stage $\log_2(N)$
Number of groups	1	2	4	...	$N/2$
Butterflies per group	$N/2$	$N/4$	$N/8$...	1
Increment exponent twiddle factors	1	2	4	...	$N/2$

We may also construct an algorithm with decimation in time (DIT). In this case, we start by splitting the input (time) sequence, and we find that all frequency values will appear in natural order (Exercise 6.10, p. 467).

The necessary index transform for index 41, for an radix-2 and radix-4 algorithm, is shown in Fig. 6.14. For a radix-2 algorithm, a reversing of the bit sequence, a *bitreverse*, is necessary. For a radix-4 algorithm we must first build “digits” of two bits, and then reverse the order of these digits. This operation is called *digitreverse*.

**Fig. 6.14.** Bitreverse and digitreverse

Radix-2 Cooley–Tukey Algorithm Implementation

A radix-2 FFT can be efficiently implemented using a butterfly processor which includes, besides the butterfly itself, an additional complex multiplier for the twiddle factors.

A radix-2 butterfly processor consists of a complex adder, a complex subtraction, and a complex multiplier for the twiddle factors. The complex multiplication with the twiddle factor is often implemented with four real multiplications and two add/subtract operations. However, it is also possible to build the complex multiplier with only three real multiplications and three

add/subtract operations, because one operand is precomputed. The algorithm works as follows:

Algorithm 6.10: Efficient Complex Multiplier

The complex twiddle factor multiplication $R + jI = (X + jY) \times (C + jS)$ can be simplified, because C and S are precomputed and stored in a table. It is therefore also possible to store the following three coefficients:

$$C, \quad C + S, \quad \text{and} \quad C - S. \quad (6.29)$$

With these three precomputed factors we first compute

$$E = X - Y, \quad \text{and then} \quad Z = C \times E = C \times (X - Y). \quad (6.30)$$

We can then compute the final product using

$$R = (C - S) \times Y + Z \quad (6.31)$$

$$I = (C + S) \times X - Z. \quad (6.32)$$

To check:

$$\begin{aligned} R &= (C - S)Y + C(X - Y) \\ &= CY - SY + CX - CY = CX - SY \checkmark \end{aligned}$$

$$\begin{aligned} I &= (C + S)X - C(X - Y) \\ &= CX + SX - CX + CY = CY + SX. \checkmark \end{aligned}$$

The algorithm uses three multiplications, one addition, and two subtractions, at the cost of an additional third table. The downside is that the worst case path is increased by one addition operation.

Cooley–Tukey Length-256 FFT Algorithm Implementation

Now we have all the data together to build a full size FFT. We follow the scheme from Table 6.3 for each stage of the FFT and develop the code to update the butterfly data, increments in twiddle factors, dual nodes, and group sizes. Overall we should expect a simulation as shown in Fig. 6.15. First the data are loaded in the FFT machine and then each stage is computed. At the end the data run through the bitreverse and the `fft_valid` flag indicates that the FFT data are forwarded to the output ports. Overall we also see the increment of the twiddle factor angle `dw` increasing with a factor 2 in each stage. The index `k2` shows the butterflies per group decreasing by a factor of 2.

Example 6.11: 256-point FFT in HDL

The following VHDL code² implements the 256-point decimation in frequency FFT:

```
-- Generic 256 point DIF FFT algorithm using a register
-- array for data and coefficients
```

² The equivalent Verilog code `fft256.v` for this example can be found in Appendix A on page 854. Synthesis results are shown in Appendix B on page 881.

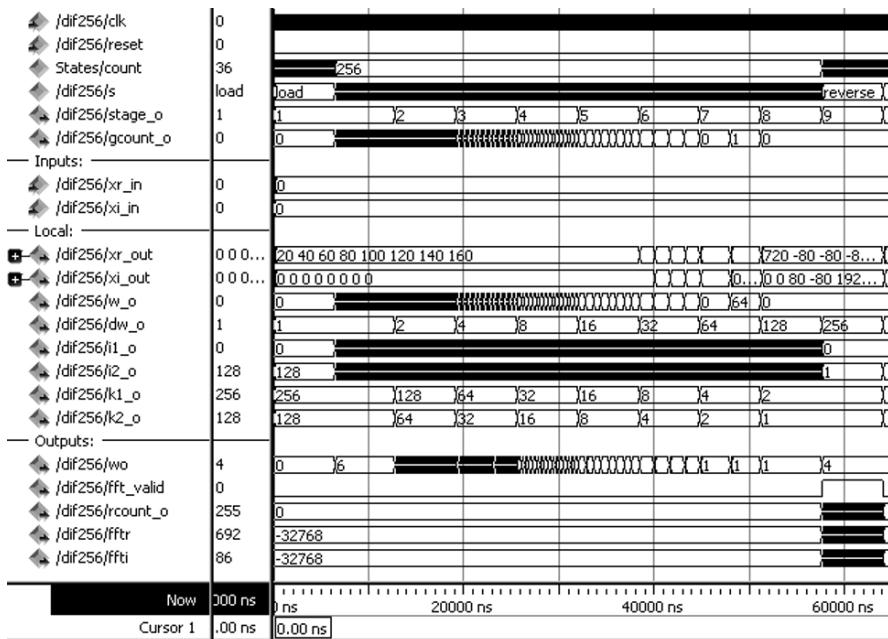


Fig. 6.15. Overall MODELSIM simulation of the 256-point FFT block

```

PACKAGE n_bits_int IS           -- User defined types
  SUBTYPE U9 IS INTEGER RANGE 0 TO 2**9-1;
  SUBTYPE S16 IS INTEGER RANGE -2**15 TO 2**15-1;
  SUBTYPE S32 IS INTEGER RANGE -2147483647 TO 2147483647;
  TYPE ARRAY0_7S16 IS ARRAY (0 TO 7) of S16;
  TYPE ARRAY0_255S16 IS ARRAY (0 TO 255) of S16;
  TYPE ARRAY0_127S16 IS ARRAY (0 TO 127) of S16;
  TYPE STATE_TYPE IS
    (start, load, calc, update, reverse, done);
END n_bits_int;

LIBRARY work; USE work.n_bits_int.ALL;

LIBRARY ieee; USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-----
ENTITY fft256 IS           -----> Interface
  PORT (clk, reset : IN STD_LOGIC; -- Clock and reset
        xr_in, xi_in : IN S16; -- Real and imag. input
        fft_valid : OUT STD_LOGIC; -- FFT output is valid
        fptr, ftti : OUT S16; -- Real and imag. output
        rcount_o : OUT U9; -- Bitreverse index counter
        xr_out, xi_out : OUT ARRAY0_7S16; -- First 8 reg. files
        stage_o, gcount_o : OUT U9; -- Stage and group count
        i1_o, i2_o : OUT U9; -- (Dual) data index

```

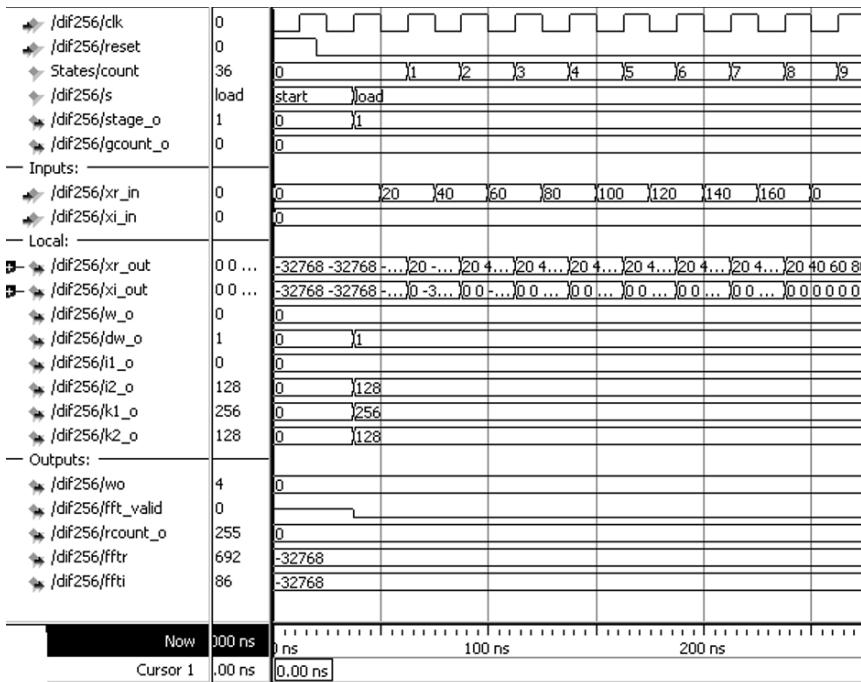


Fig. 6.16. FFT simulation input data. Start of the input frame

```

k1_o, k2_o : OUT U9; -- Index offset
w_o, dw_o : OUT U9; -- Cos/Sin (increment) angle
wo : OUT U9);      -- Decision tree location loop FSM
END fft256;
-----
ARCHITECTURE fpga OF fft256 IS
  SIGNAL s      : STATE_TYPE; -- State machine variable
  CONSTANT N : U9 := 256; -- Number of points
  CONSTANT ldN : U9 := 8; -- Log_2 number of points
  -- Register array for 16 bit precision:
  SIGNAL xr, xi : ARRAY0_255S16;
  SIGNAL w : U9 := 0;
  -- sine and cosine coefficient arrays
  -----
  CONSTANT cos_rom : ARRAY0_127S16 := (16384,16379,16364,16340
,16305,16261,16207,16143,16069,15986,15893,15791,15679,
15557,15426,15286,15137,14978,14811,14635,14449,14256,
14053,13842,13623,13395,13160,12916,12665,12406,12140,
11866,11585,11297,11003,10702,10394,10080,9760,9434,9102,
8765,8423,8076,7723,7366,7005,6639,6270,5897,5520,5139,
4756,4370,3981,3590,3196,2801,2404,2006,1606,1205,804,402

```

```

,0,-402,-804,-1205,-1606,-2006,-2404,-2801,-3196,-3590,
-3981,-4370,-4756,-5139,-5520,-5897,-6270,-6639,-7005,
-7366,-7723,-8076,-8423,-8765,-9102,-9434,-9760,-10080,
-10394,-10702,-11003,-11297,-11585,-11866,-12140,-12406,
-12665,-12916,-13160,-13395,-13623,-13842,-14053,-14256,
-14449,-14635,-14811,-14978,-15137,-15286,-15426,-15557,
-15679,-15791,-15893,-15986,-16069,-16143,-16207,-16261,
-16305,-16340,-16364,-16379);

CONSTANT sin_rom : ARRAY0_127S16 := (0,402,804,1205,1606,
2006,2404,2801,3196,3590,3981,4370,4756,5139,5520,5897,
6270,6639,7005,7366,7723,8076,8423,8765,9102,9434,9760,
10080,10394,10702,11003,11297,11585,11866,12140,12406,
12665,12916,13160,13395,13623,13842,14053,14256,14449,
14635,14811,14978,15137,15286,15426,15557,15679,15791,
15893,15986,16069,16143,16207,16261,16305,16340,16364,
16379,16384,16379,16364,16340,16305,16261,16207,16143,
16069,15986,15893,15791,15679,15557,15426,15286,15137,
14978,14811,14635,14449,14256,14053,13842,13623,13395,
13160,12916,12665,12406,12140,11866,11585,11297,11003,
10702,10394,10080,9760,9434,9102,8765,8423,8076,7723,
7366,7005,6639,6270,5897,5520,5139,4756,4370,3981,3590,
3196,2801,2404,2006,1606,1205,804,402);

SIGNAL sin , cos : S16;
BEGIN

sin_read: PROCESS (clk)
BEGIN
    IF falling_edge(clk) THEN
        sin <= sin_rom(w); -- Read from ROM
    END IF;
END PROCESS;

cos_read: PROCESS (clk)
BEGIN
    IF falling_edge(clk) THEN
        cos <= cos_rom(w); -- Read from ROM
    END IF;
END PROCESS;

States: PROCESS(clk, reset, w)----> FFT in behavioral style
VARIABLE i1, i2, gcount, k1, k2 : U9 := 0;
VARIABLE stage, dw, count, rcount : U9 := 0;
VARIABLE tr, ti : S16 := 0;
VARIABLE slv, rslv : STD_LOGIC_VECTOR(0 TO 1dN-1);
BEGIN
    IF reset = '1' THEN                      -- Asynchronous reset
        s <= start;
    ELSIF rising_edge(clk) THEN
        CASE s IS                         -- Next State assignments
        WHEN start =>
            s <= load; count := 0;
        WHEN ...
        END CASE;
    END IF;
END;

```

```

gcount := 0; stage:= 1; i1:=0; i2 := N/2; k1:=N;
k2:=N/2; dw := 1; fft_valid <= '0';
WHEN load =>           -- Read in all data from I/O ports
    xr(count) <= xr_in; xi(count) <= xi_in;
    count := count + 1;
    IF count = N THEN   s <= calc;
    ELSE                 s <= load;
    END IF;
WHEN calc =>            -- Do the butterfly computation
    tr := xr(i1) - xr(i2);
    xr(i1) <= xr(i1) + xr(i2);
    ti := xi(i1) - xi(i2);
    xi(i1) <= xi(i1) + xi(i2);
    xr(i2) <= (cos * tr + sin * ti)/2**14;
    xi(i2) <= (cos * ti - sin * tr)/2**14;
    s <= update;
WHEN update =>          -- All counters and pointers
    s <= calc; -- By default do next butterfly
    i1 := i1 + k1; -- Next butterfly in group
    i2 := i1 + k2;
    wo <= 1;
    IF i1 >= N-1 THEN -- All butterflies done in group?
        gcount := gcount + 1;
        i1 := gcount;
        i2 := i1 + k2;
        wo <= 2;
        IF gcount >= k2 THEN-- All groups done in stages?
            gcount := 0; i1 := 0; i2 := k2;
            dw := dw * 2;
            stage := stage + 1;
            wo <= 3;
            IF stage > ldN THEN -- All stages done
                s <= reverse;
                count := 0;
                wo <= 4;
            ELSE -- Start new stage
                k1 := k2; k2 := k2/2;
                i1 := 0; i2 := k2;
                w <= 0;
                wo <= 5;
            END IF;
        ELSE -- Start new group
            i1 := gcount; i2 := i1 + k2;
            w <= w + dw;
            wo <= 6;
        END IF;
    END IF;
WHEN reverse =>          -- Apply bitreverse
    fft_valid <= '1';
    slv := CONV_STD_LOGIC_VECTOR(count, ldN);
    FOR i IN 0 TO ldN-1 LOOP
        rslv(i) := slv(ldN-i-1);
    END LOOP;

```

```

rcount := CONV_INTEGER('0' & rslv);
fftr <= xr(rcount); ffti <= xi(rcount);
count := count + 1;
IF count >= N THEN  s <= done;
ELSE                  s <= reverse;
END IF;
WHEN done =>      -- Output of results
    s <= start;     -- Start next cycle
    END CASE;
END IF;
i1_o<=i1;   -- Provide some test signals as outputs
i2_o<=i2;
stage_o<=stage;
gcount_o <= gcount;
k1_o <= k1;
k2_o<=k2;
w_o<=w;
dw_o<=dw;
rcount_o <= rcount;
END PROCESS States;

Rk: FOR k IN 0 TO 7 GENERATE -- Show first 8
    xr_out(k) <= xr(k);           -- register values
    xi_out(k) <= xi(k);
END GENERATE;

END fpga;

```

The design starts with user type specification followed by the I/O port description in the ENTITY part. The ARCHITECTURE section starts with a few signal definitions and then the 128 values for the cos and sin tables. The FFT is implemented with one PROCESS statement only. The first **start** state of the state machine initializes major variables and loop counters. It is followed by the **load** state where all real and imaginary data are read into an internal 256×16 register file. The **calc** and **update** state then alternate for many clock cycles. In the **calc** phase the new butterfly is computed. Note that we use the conventional four multiply and two add types to reduce the worst case delay path. In the **update** state the main loop counters and data index are updated. We see the three loop hierarchies: butterfly, groups, and stages. We then proceed to the **reverse** state where we compute the bitreverse of the data. Immediately within the **reverse** state we forward the FFT values to the output ports and at the same time set the **fft_valid** flag to one to indicate that valid FFT values appear at the output ports.

The design uses 34,340 LEs, eight embedded multipliers, and has an **Fmax**=31.12 MHz registered performance using the TimeQuest slow 85C model. The used LEs of the VHDL design will be reduced for an optimization goal **Area**, since then the sine and cosine LUTs will be synthesized to embedded M9K blocks. For optimization **Speed** the VHDL LUTs are synthesized to LEs. In Verilog the use of the **\$readmemh()** function means that, for **Speed** and **Area** optimization, both times two embedded M9Ks are synthesized; see synthesis results on p. 881.

Figure 6.16 displays simulation start results using MODELSIM for a triangular input sequence $x[n] = \{20, 40, 60, 80, 100, 120, 140, 160, 0 \dots\}$. Only the first

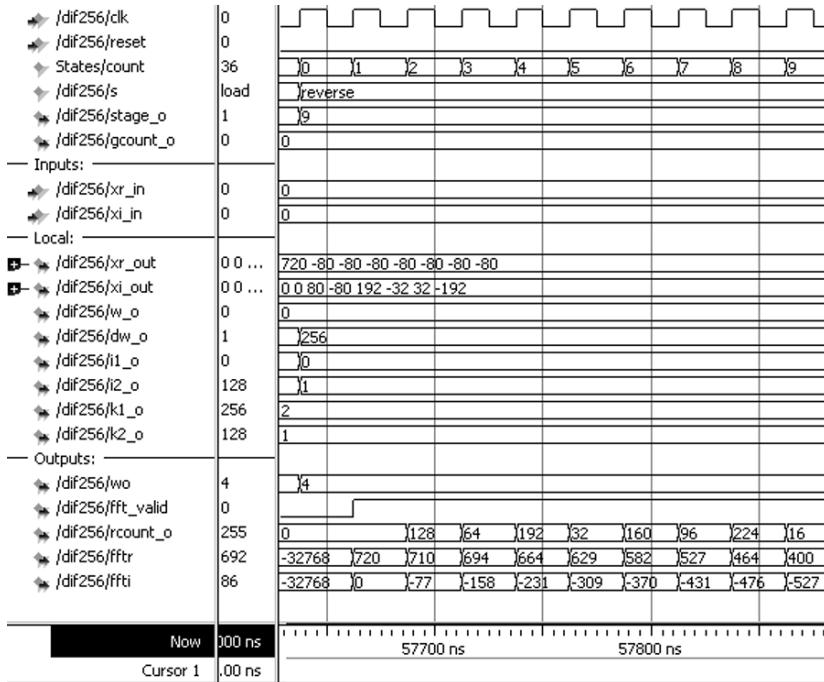


Fig. 6.17. FFT core simulation output results. Start of the output frame

eight values are nonzero. The test sequence can be generated in MATLAB as follows:

```
x=[(1:8)*20,zeros(1,248)];
Y=fft(x);
```

with the following instructions we quantize and list the first ten samples as in the MODELSIM simulation:

```
sprintf('%d ',real(round(Y(1:9))))
sprintf('%d ',imag(round(Y(1:9))))
```

and the (expected) test data will be

```
720 714 698 671 634 587 532 471 403 ...
0 -82 -163 -240 -313 -380 -439 -490 -532 ...
```

Finally, at 57.7 μ s, we see from Fig. 6.17 the DC value, i.e., $\sum x_i = 720$ followed by the other FFT values that match our MATLAB test data if we allow some quantization errors.

6.11

We may think of several ideas regarding how to improve the design. The most effective in terms of LEs will be if we replace the three large register files by embedded memory blocks. This will however require the use of additional states in the FFT machine since the M9K can only be used as synchronous memory.

6.2.2 The Good–Thomas FFT Algorithm

The index transform suggested by Good [198] and Thomas [199] transforms a DFT of length $N = N_1 N_2$ into an “actual” two-dimensional DFT, i.e., there are no twiddle factors as in the Cooley–Tukey FFT. The price we pay for the twiddle factor free flow is that the factors must be coprime (i.e., $\gcd(N_k, N_l) = 1$ for $k \neq l$), and we have a somewhat more complicated index mapping, as long as the index computation is done “online” and no precomputed tables are used for the index mapping.

If we try to eliminate the twiddle factors introduced through the index mapping of n and k according to (6.17) and (6.19), respectively, it follows from

$$\begin{aligned} W_N^{nk} &= W_N^{(An_1+Bn_2)(Ck_1+Dk_2)} \\ &= W_N^{ACn_1k_1+ADn_1k_2+BCk_1n_2+BDn_2k_2} \\ &= W_N^{N_2n_1k_1} W_N^{N_1k_2n_2} = W_{N_1}^{n_1k_1} W_{N_2}^{k_2n_2}, \end{aligned} \quad (6.33)$$

that we must fulfill all the following necessary conditions at the same time:

$$\langle AD \rangle_N = \langle BC \rangle_N = 0 \quad (6.34)$$

$$\langle AC \rangle_N = N_2 \quad (6.35)$$

$$\langle BD \rangle_N = N_1. \quad (6.36)$$

The mapping suggested by Good [198] and Thomas [199] fulfills this condition and is given by

$$A = N_2 \quad B = N_1 \quad C = N_2 \langle N_2^{-1} \rangle_{N_1} \quad D = N_1 \langle N_1^{-1} \rangle_{N_2}. \quad (6.37)$$

To check: Because the factors AD and BC both include the factor $N_1 N_2 = N$, it follows that (6.34) is checked. With $\gcd(N_1, N_2) = 1$ and a theorem due to Euler, we can write the inverse as $N_2^{-1} \bmod N_1 = N_2^{\phi(N_1)-1} \bmod N_1$ where ϕ is the Euler totient function. The condition (6.35) can now be rewritten as

$$\langle AC \rangle_N = \langle N_2 N_2 \langle N_2^{\phi(N_1)-1} \rangle_{N_1} \rangle_N. \quad (6.38)$$

We can now solve the inner modulo reduction, and it follows with $\nu \in \mathbb{Z}$ and $\nu N_1 N_2 \bmod N = 0$ finally

$$\langle AC \rangle_N = \langle N_2 N_2 (N_2^{\phi(N_1)-1} + \nu N_1) \rangle_N = N_2. \quad (6.39)$$

The same argument can be applied for the condition (6.36), and we have shown that all three conditions from (6.34)–(6.36) are fulfilled if the Good–Thomas mapping (6.37) is used. \square

In conclusion, we can now define the following theorem

Theorem 6.12: Good–Thomas Index Mapping

The index mapping suggested by Good and Thomas for n is

$$n = N_2 n_1 + N_1 n_2 \bmod N \quad \begin{cases} 0 \leq n_1 \leq N_1 - 1 \\ 0 \leq n_2 \leq N_2 - 1 \end{cases} \quad (6.40)$$

and as index mapping for k results

$$k = N_2 \langle N_2^{-1} \rangle_{N_1} k_1 + N_1 \langle N_1^{-1} \rangle_{N_2} k_2 \bmod N \quad \begin{cases} 0 \leq k_1 \leq N_1 - 1 \\ 0 \leq k_2 \leq N_2 - 1 \end{cases} . \quad (6.41)$$

The transform from (6.41) is identical to the Chinese remainder theorem 2.13 (p. 71). It follows, therefore, that k_1 and k_2 can simply be computed via a modulo reduction, i.e., $k_l = k \bmod N_l$.

If we now substitute the Good–Thomas index map in the equation for the DFT matrix (6.16), it follows that

$$X[k_1, k_2] = \sum_{n_2=0}^{N_2-1} W_{N_2}^{n_2 k_2} \underbrace{\left(\sum_{n_1=0}^{N_1-1} x[n_1, n_2] W_{N_1}^{n_1 k_1} \right)}_{\text{N}_1\text{-point transform}} \quad (6.42)$$

$$= \underbrace{\sum_{n_2=0}^{N_2-1} W_{N_2}^{n_2 k_2} \bar{x}[n_2, k_1]}_{\text{N}_2\text{-point transform}}, \quad (6.43)$$

i.e., as claimed at the beginning, it is an “actual” two-dimensional DFT transform without the twiddle factor introduced by the mapping suggested by Cooley and Tukey. It follows that the Good–Thomas algorithm, although similar to the Cooley–Tukey Algorithm 6.8, has a different index mapping and no twiddle factors.

Algorithm 6.13: Good–Thomas FFT Algorithm

An $N = N_1 N_2$ -point DFT can be computed according to the following steps:

- 1) Index transform of the input sequence, according to (6.40).
- 2) Computation of N_2 DFTs of length N_1 .
- 3) Computation of N_1 DFTs of length N_2 .
- 4) Index transform of the output sequence, according to (6.41).

An $N = 12$ transform shown in the following example demonstrates the steps.

Example 6.14: Good–Thomas FFT Algorithm for $N = 12$

Suppose we have $N_1 = 4$ and $N_2 = 3$. Then a mapping for the input index according to $n = 3n_1 + 4n_2 \bmod 12$, and $k = 9k_1 + 4k_2 \bmod 12$ for the output index results, and we can compute the following index mapping tables:

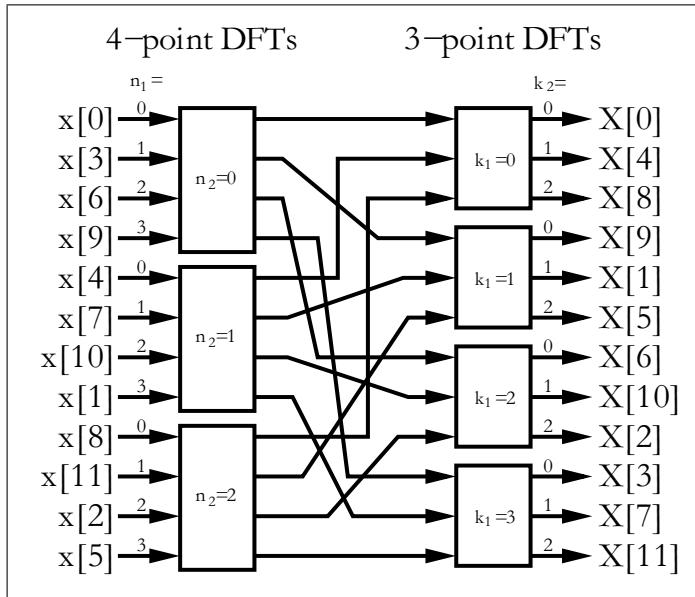


Fig. 6.18. Good–Thomas FFT for $N = 12$

n_2	0	n_1	2	3	k_2	0	k_1	2	3
0	$x[0]$	$x[3]$	$x[6]$	$x[9]$	0	$X[0]$	$X[9]$	$X[6]$	$X[3]$
1	$x[4]$	$x[7]$	$x[10]$	$x[1]$	1	$X[4]$	$X[1]$	$X[10]$	$X[7]$
2	$x[8]$	$x[11]$	$x[2]$	$x[5]$	2	$X[8]$	$X[5]$	$X[2]$	$X[11]$

Using these index transforms we can construct the signal flow graph shown in Fig. 6.18. We realize that the first stage has three DFTs each having four points and the second stage four DFTs each of length 3. Multiplication by twiddle factors between the stages is not necessary. 6.14

6.2.3 The Winograd FFT Algorithm

The Winograd FFT algorithm [124] is based on the observation that the inverse DFT matrix (6.4) (without prefactor N^{-1}) of dimension $N_1 \times N_2$, with $\gcd(N_1, N_2) = 1$, i.e.,

$$x[n] = \sum_{k=0}^{N-1} X[k] W_N^{-nk} \quad (6.44)$$

$$\mathbf{x} = \mathbf{W}_N^* \mathbf{X} \quad (6.45)$$

can be rewritten using the *Kronecker product*³ with two quadratic IDFT matrices each, with dimension N_1 and N_2 , respectively. As with the index mapping for the Good–Thomas algorithm, we must write the indices of $X[k]$ and $x[n]$ in a two-dimensional scheme and then read out the indices row by row. The following example for $N = 12$ demonstrates the steps.

Example 6.15: IDFT using Kronecker Product for $N = 12$

Let $N_1 = 4$ and $N_2 = 3$. Then we have the output index transform $k = 9k_1 + 4k_2 \bmod 12$ according to the Good–Thomas index mapping:

$$\left[\begin{array}{c} X[0] \\ X[1] \\ X[2] \\ X[3] \\ X[4] \\ X[5] \\ X[6] \\ X[7] \\ X[8] \\ X[9] \\ X[10] \\ X[11] \end{array} \right] \xrightarrow{k_2 \left| \begin{array}{cccc} & & k_1 & \\ 0 & 1 & 2 & 3 \end{array} \right.} \left[\begin{array}{c} X[0] \\ X[9] \\ X[6] \\ X[3] \\ X[4] \\ X[1] \\ X[10] \\ X[7] \\ X[8] \\ X[5] \\ X[2] \\ X[11] \end{array} \right].$$

We can now construct a length-12 IDFT with

$$\left[\begin{array}{c} x[0] \\ x[9] \\ x[6] \\ x[3] \\ x[4] \\ x[1] \\ x[10] \\ x[7] \\ x[8] \\ x[5] \\ x[2] \\ x[11] \end{array} \right] = \left[\begin{array}{ccc} W_{12}^0 & W_{12}^0 & W_{12}^0 \\ W_{12}^0 & W_{12}^{-4} & W_{12}^{-8} \\ W_{12}^0 & W_{12}^{-8} & W_{12}^{-4} \end{array} \right] \otimes \left[\begin{array}{cccc} W_{12}^0 & W_{12}^0 & W_{12}^0 & W_{12}^0 \\ W_{12}^0 & W_{12}^{-3} & W_{12}^{-6} & W_{12}^{-9} \\ W_{12}^0 & W_{12}^{-6} & W_{12}^0 & W_{12}^{-6} \\ W_{12}^0 & W_{12}^{-9} & W_{12}^{-6} & W_{12}^{-3} \end{array} \right] \left[\begin{array}{c} X[0] \\ X[9] \\ X[6] \\ X[3] \\ X[4] \\ X[1] \\ X[10] \\ X[7] \\ X[8] \\ X[5] \\ X[2] \\ X[11] \end{array} \right].$$
6.15

So far we have used the Kronecker product to (re)define the IDFT. Using the short-hand notation $\tilde{\mathbf{x}}$ for the permuted sequence \mathbf{x} , we may use the following matrix/vector notation:

³ A Kronecker product is defined by

$$\begin{aligned} \mathbf{A} \otimes \mathbf{B} &= [a[i,j]]\mathbf{B} \\ &= \begin{bmatrix} a[0,0]\mathbf{B} & \cdots & a[0,L-1]\mathbf{B} \\ \vdots & & \vdots \\ a[K-1,0]\mathbf{B} & \cdots & a[K-1,L-1]\mathbf{B} \end{bmatrix} \end{aligned}$$

where \mathbf{A} is a $K \times L$ matrix.

$$\tilde{\mathbf{x}} = \mathbf{W}_{N_1} \otimes \mathbf{W}_{N_2} \tilde{\mathbf{X}}. \quad (6.46)$$

For these short DFTs we now use the Winograd DFT Algorithm 6.7 (p. 434), i.e.,

$$\mathbf{W}_{N_l} = \mathbf{C}_l \times \mathbf{B}_l \times \mathbf{A}_l, \quad (6.47)$$

where \mathbf{A}_l incorporate the input additions, \mathbf{B}_l is a diagonal matrix with the Fourier coefficients, and \mathbf{C}_l includes the output additions. If we now substitute (6.47) into (6.46), and use the fact that we can change the sequence of matrix multiplications and Kronecker product computation (see for instance [5, App. D]), we get

$$\begin{aligned} \mathbf{W}_{N_1} \otimes \mathbf{W}_{N_2} &= (\mathbf{C}_1 \times \mathbf{B}_1 \times \mathbf{A}_1) \otimes (\mathbf{C}_2 \times \mathbf{B}_2 \times \mathbf{A}_2) \\ &= (\mathbf{C}_1 \otimes \mathbf{C}_2)(\mathbf{B}_1 \otimes \mathbf{B}_2)(\mathbf{A}_1 \otimes \mathbf{A}_2). \end{aligned} \quad (6.48)$$

Because the matrices \mathbf{A}_l and \mathbf{C}_l are simple addition matrices, the same applies for its Kronecker products, $\mathbf{A}_1 \otimes \mathbf{A}_2$ and $\mathbf{C}_1 \otimes \mathbf{C}_2$. The Kronecker product of two quadratic diagonal matrices of dimension N_1 and N_2 , respectively, obviously also gives a diagonal matrix of dimension $N_1 N_2$. The total number of necessary multiplications is therefore identical to the number of diagonal elements of $\mathbf{B} = \mathbf{B}_1 \otimes \mathbf{B}_2$, i.e., $M_1 M_2$, if M_1 and M_2 , respectively, are the number of multiplications used to compute the smaller Winograd DFTs according to Table 6.2 (p. 434).

We can now combine the different steps to construct a Winograd FFT.

Theorem 6.16: **Winograd FFT Design**

A $N = N_1 N_2$ -point transform with coprimes N_1 and N_2 can be constructed as follows:

- 1) Index transform of the input sequence according to the Good–Thomas mapping (6.40), followed by a row read of the indices.
- 2) Factorization of the DFT matrix using the Kronecker product.
- 3) Substitute the length N_1 and N_2 DFT matrices through the Winograd DFT algorithm.
- 4) Centralize the multiplications.

After successful construction of the Winograd FFT algorithm, we can compute the Winograd FFT using the following three steps:

Theorem 6.17: **Winograd FFT Algorithm**

- 1) Compute the preadditions \mathbf{A}_1 and \mathbf{A}_2 .
- 2) Compute $M_1 M_2$ multiplications according to the matrix $\mathbf{B}_1 \otimes \mathbf{B}_2$.
- 3) Compute postadditions according to \mathbf{C}_1 and \mathbf{C}_2 .

Let us now look at a construction of a Winograd FFT of length-12, in detail in the following example.

Example 6.18: Winograd FFT of Length 12

To build a Winograd FFT, we have, according to Theorem 6.16, to compute the necessary matrices used in the transform. For $N_1 = 3$ and $N_2 = 4$ we have the following matrices:

$$\mathbf{A}_1 \otimes \mathbf{A}_2 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \quad (6.49)$$

$$\mathbf{B}_1 \otimes \mathbf{B}_2 = \text{diag}(1, -3/2, \sqrt{3}/2) \otimes \text{diag}(1, 1, 1, -i) \quad (6.50)$$

$$\mathbf{C}_1 \otimes \mathbf{C}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & i \\ 1 & 1 & -i \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix}. \quad (6.51)$$

Combining these matrices according to (6.48) results in the Winograd FFT algorithm. Input and output additions can be realized multiplier free, and the total number of real multiplication becomes $2 \times 3 \times 4 = 24$. 6.18

So far we have used the Winograd FFT to compute the IDFT. If we now want to compute the DFT with the help of an IDFT, we can use a technique we used in (6.6) on p. 420 to compute the IDFT with help of the DFT. Using matrix/vector notation we find

$$\mathbf{x}^* = (\mathbf{W}_N^* \mathbf{X})^* \quad (6.52)$$

$$\mathbf{x}^* = \mathbf{W}_N \mathbf{X}^*, \quad (6.53)$$

if $\mathbf{W}_N = [e^{2\pi j nk/N}]$ with $n, k \in \mathbb{Z}_N$ is a DFT. The DFT can therefore be computed using the IDFT with the following steps: Compute the conjugate complex of the input sequence, transform the sequence with the IDFT algorithm, and compute the conjugate complex of the output sequence.

It is also possible to use the Kronecker product algorithms, i.e., the Winograd FFT, to compute the DFT directly. This leads to a slide-modified output index mapping, as the following example shows.

Example 6.19: A 12-point DFT can be computed using the following Kronecker product formulation:

$$\begin{bmatrix} X[0] \\ X[3] \\ X[6] \\ X[9] \\ X[4] \\ X[7] \\ X[10] \\ X[1] \\ X[8] \\ X[11] \\ X[2] \\ X[5] \end{bmatrix} = \begin{bmatrix} W_{12}^0 & W_{12}^0 & W_{12}^0 \\ W_{12}^0 & W_{12}^4 & W_{12}^8 \\ W_{12}^0 & W_{12}^8 & W_{12}^4 \end{bmatrix} \otimes \begin{bmatrix} W_{12}^0 & W_{12}^0 & W_{12}^0 & W_{12}^0 \\ W_{12}^0 & W_{12}^3 & W_{12}^6 & W_{12}^9 \\ W_{12}^0 & W_{12}^6 & W_{12}^0 & W_{12}^6 \\ W_{12}^0 & W_{12}^9 & W_{12}^6 & W_{12}^3 \end{bmatrix} \cdot \begin{bmatrix} x[0] \\ x[9] \\ x[6] \\ x[3] \\ x[4] \\ x[1] \\ x[10] \\ x[7] \\ x[8] \\ x[5] \\ x[2] \\ x[11] \end{bmatrix}. \quad (6.54)$$

The input sequence $x[n]$ can be considered to be in the order used for Good–Thomas mapping, while in the (frequency) output index mapping for $X[k]$, each first and third element are exchanged, compared with the Good–Thomas mapping.

6.19

6.2.4 Comparison of DFT and FFT Algorithms

It should now be apparent that there are many ways to implement a DFT. The choice begins with the selection of a short DFT algorithm from among those shown in Fig. 6.1 (p. 417). The short DFT can then be used to develop long DFTs, using the indexing schemes provided by Cooley–Tukey, Good–Thomas, or Winograd. A common objective in choosing an implementation is minimum multiplication complexity. This is a viable criterion when the implementation cost of multiplication is much higher compared with other operations, such as additions, data access, or index computation.

Figure 6.19 shows the number of multiplications required for various FFT lengths. It can be concluded that the Winograd FFT is most attractive, based purely on a multiply complexity criterion. In this chapter, the design of $N = 4 \times 3 = 12$ -point FFTs has been presented in several forms. A comparison of a direct, Rader prime-factor algorithms, and Winograd DFT algorithms used for the basic DFT blocks, and the three different index mappings called Good–Thomas, Cooley–Tukey, and Winograd FFT, is presented in Table 6.4.

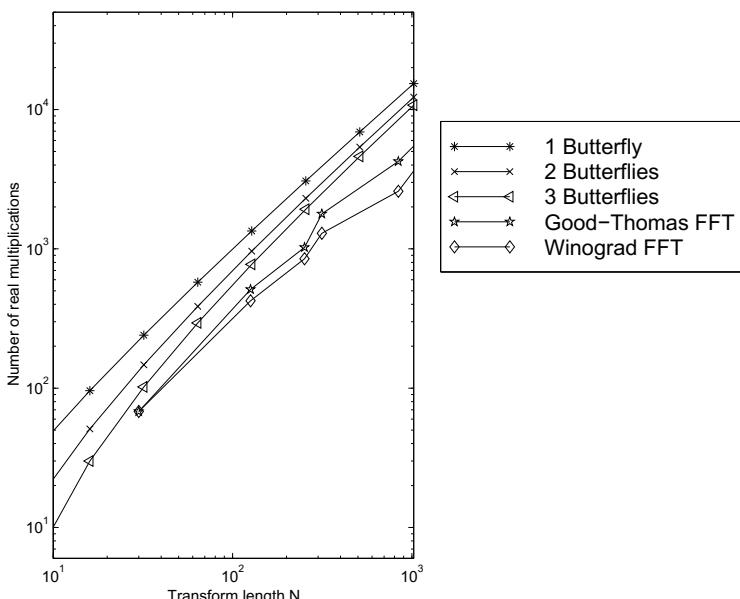


Fig. 6.19. Comparison of different FFT algorithm based on the number of necessary real multiplications

Table 6.4. Number of real multiplications for a length-12 complex input FFT algorithm (twiddle factor multiplications by W^0 are not counted). A complex multiplication is assumed to use four real multiplications

Index mapping			
DFT Method	Good–Thomas Fig. 6.18 p. 451	Cooley–Tukey Fig. 6.2 p. 437	Winograd Example 6.15 p. 452
Direct	$4 \times 12^2 = 4 \times 144 = 576$		
RPFA	$4(3(4 - 1)^2 + 4(3 - 1)^2) = 172$	$4(43 + 6) = 196$	–
WFTA	$3 \times 0 \times 2 + 4 \times 2 \times 2 = 16$	$16 + 4 \times 6 = 40$	$2 \times 3 \times 4 = 24$

Besides the number of multiplications, other constraints must be considered, such as possible transform lengths, number of additions, index computation overhead, coefficient or data memory size, and run-time code length. In many cases, the Cooley–Tukey method provides the best overall solution, as suggested by Table 6.5.

Some of the published FPGA realizations are summarized in Table 6.6. The design by Goslin [184] is based on a radix-2 FFT, in which the butterflies have been realized using distributed arithmetic, discussed in Chap. 2. The design by Dandalis et al. [200], is based on an approximation of the DFT using the so-called arithmetic Fourier transform. The ERNS FFT, from Meyer-Bäse et al. [201], uses the Rader algorithm in combination with the number theoretic transform.

With FPGAs reaching complexities of more than 1M gates today, full integration of an FFT on a single FPGA is viable. Because the design of such an FFT block is labor intensive, it most often makes sense to utilize commercially available “intellectual property” (IP) blocks (sometimes also called “virtual components” VCs). See, for instance, the IP partnership programs at www.xilinx.com or www.altera.com. The majority of the commercially available designs are based on radix-2 or radix-4.

6.2.5 IP Core FFT Design

Altera and Xilinx offer FFT generators, since this is, besides the FIR filter, one of the most often used intellectual property (IP) blocks. For an introduction to IP blocks see Sect. 1.4.4, p. 40. Xilinx has some free fixed-length and bit width hard core [202], but usually the FFT parameterized cores have to be purchased from the FPGA vendors for a (reasonable) licence fee.

Let us have a look at the 256-point FFT generation that we discussed before, see Example 6.11, p. 442, for a radix-2 DIF FFT code. But this

Table 6.5. Important properties for FFT algorithms of length $N = \prod N_k$

Property	Cooley–Tukey	Good–Thomas	Winograd
Any transform Length	yes	no $\gcd(N_k, N_l) = 1$	
Maximum order of W	N		$\max(N_k)$
Twiddle factors needed	yes	no	no
# Multiplications	bad	fair	best
# Additions	fair	fair	fair
# Index computation effort	best	fair	bad
Data in-place	yes	yes	no
Implementation advantages	small butterfly processor	can use RPFA, fast, simple FIR array	small size for full parallel, medium-size FFT (< 50)

Table 6.6. Comparison of some FPGA FFT implementations [5]

Name	Data type	FFT type	N -point FFT time	Clock rate P	Internal RAM/ROM	Design aim/source
Xilinx FPGA	8-bit	Radix=2 FFT	$N = 256$ 102.4 μ s	70 MHz 4.8 W @3.3 V	No	573 CLBs [184]
Xilinx FPGA	16-bit	AFT	$N = 256$ 82.48 μ s 42.08 μ s	50 MHz 15.6 W @ 3.3 V 29.5 W	No	[200] 2602 CLBs 4922 CLBs
Xilinx FPGA ERNS-NTT	12.7 bit	FFT using NTT	$N = 97$ 9.24 μ s	26 MHz 3.5 W @3.3 V	No	1178 CLBs [201]

time we use the Altera FFT compiler [203] to build the FFT, including the FSM to control the processing. The Altera FFT MegaCore function is a high performance, highly parameterizable FFT processor optimized for Altera devices. Stratix and Cyclone IV devices are supported but no mature devices from the APEX or Flex family. The FFT function implements a radix-2/4 decimation-in-frequency (DIF) FFT algorithm for transform lengths of 2^S , where $6 \leq S \leq 16$. Internally a block-floating-point architecture is used to maximize signal dynamic range in the transform calculation. You can use the

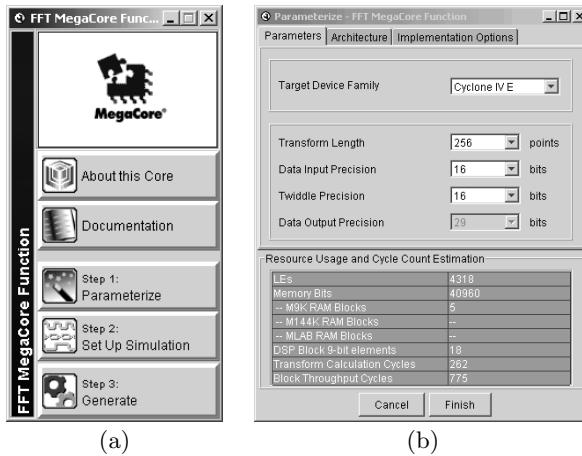


Fig. 6.20. IP design of an FFT **(a)** IP toolbench. **(b)** Coefficient specification

IP toolbench MEGA WIZARD design environment to specify a variety of FFT architectures, including $4 \times 2+$ and $3 \times 5+$ butterfly architectures and different parallel architectures. The FFT compiler includes a coefficient generator for the twiddle factors that are stored in M9K blocks.

Example 6.20: Length-256 FFT IP Generation

To start the Altera FFT compiler we select **MegaWizard Plug-In Manager** under the **Tools** menu, and the library selection window (see Fig. 1.22, p. 43) will pop up. The FFT generator can be found under **DSP→Transform**. You need to specify a design name for the core and we can then proceed to the **ToolBench**; see Fig. 6.20a. We first select **Parameters** and choose as the FFT length 256, and set the data and coefficient precision to 16 bits. We then have a choice of different architecture: **Streaming**, **Buffered Burst**, and **Burst**. The different architectures use more or fewer additional buffers to allow block-by-block processing that requires, in the case of the **Streaming** architecture, no additional clock cycles. Every 256 cycles, we submit a new data set to the FFT core and, after some processing, get a new 256-point data set. With the $3 \times 5+$ selection in the **Implementation Options** the logic resource estimation for the Cyclone IV family will be as follows:

Resource	Streaming	Buffered burst	Burst
LEs	4581	4638	4318
M9K	11	9	5
M144K RAM	0	0	0
MLAB	0	0	0
DSP block 9-bit	18	18	18
Transform calculation cycles	256	258	262
Block throughput cycles	256	331	775

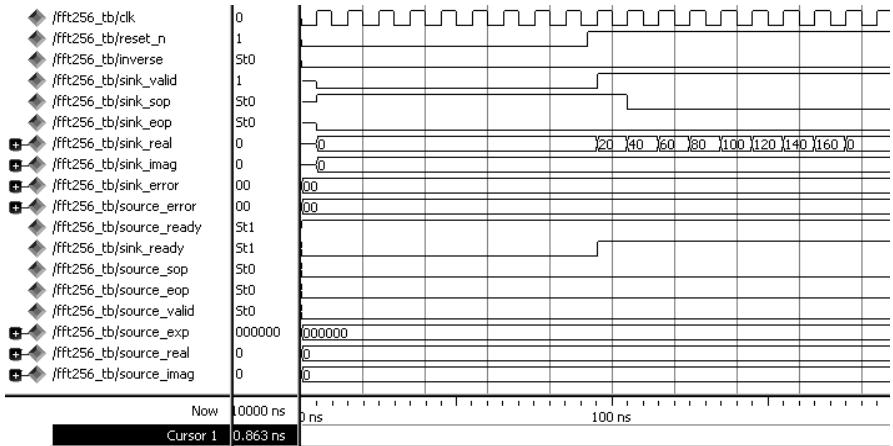


Fig. 6.21. Quartus II simulation of the IP FFT block initialization steps

Step 2 from the toolbench will generate a simulation model that is required for the MODELSIM simulation. We proceed with step 3, and the generation of the VHDL code and all supporting files follows. The coefficient files for the twiddle factor, along with MATLAB test benches and MODELTECH simulation files and scripts, are generated within a few seconds. These files are listed in Table 6.7. We see that not only are the VHDL and Verilog files generated along with their component file, but MATLAB (bit accurate) and MODELSIM (cycle accurate) test vectors are provided to enable an easy verification path. In order to match the simulation from the DIF Example 6.11, p. 442 we need to put together our own test bench and replace the `fft256_real_input.txt` and `fft256_imag_input.txt` data files with our test data. As test data we use a short triangular sequence generated in MATLAB as follows:

```
x=[(1:8)*20,zeros(1,248+3*256)]; % Total 1024 samples
fid = fopen('fft256_real_umb.txt','w');
fprintf(fid,'%d\r\n',x); fclose(fid);
Y=fft(x);
```

Similar for the imaginary we store an zero vector. With the following instruction we quantize and list the first ten samples scaled by 2^{-3} as in the MODELSIM simulation:

```
sprintf('%d ',real(round(Y(1:10)*2^-3)))
sprintf('%d ',imag(round(Y(1:10)*2^-3)))
```

and the (expected) test data will be

```
90 89 87 84 79 73 67 59 50 41 ...      (real)
0 -10 -20 -30 -39 -47 -55 -61 -66 -70 ...      (imag)
```

The simulation of the FFT block is shown in Figs. 6.21 and 6.22. We see that the processing works in several steps. After `reset` is low we set the data valid signal `..._valid` from the sink. We then set the signal processing start flag `sink_sop` to low after one clock cycle. At the same time we apply the first input data (i.e., value 20 in our test) to the FFT block followed by the next data in each clock cycles. After 256 clock cycles all input data are processed. Since the FFT uses Steaming mode a total latency of one extra

Table 6.7. IP files generation for FFT core

File	Description
fft256.vhd	A MegaCore function variation file, which defines a top-level VHDL description of the custom MegaCore function
fft256.	instantiation file
fft256.cmp	A VHDL component declaration for the MegaCore function variation
fft256.bsf	Quartus II symbol file to be used in the Quartus II block diagram editor
fft256.vho	Functional model used by the MODELSIM simulation
fft_tb.vhd	Test bench used by the MODELSIM simulation
fft256_model.m	This file provides a MATLAB simulation model for the customized FFT
fft256_tb.m	This file provides a MATLAB test bench for the customized FFT
*.txt	Two text files with random real and imaginary input data
*.hex	Six sin/cos twiddle coefficient tables
fft256_nativelink.tcl	A Tcl script to setup NativeLink in the Quartus II software.
fft256.qip	Contains Quartus II project information for your MegaCore function variation.
fft256_syn.v	A timing and resource estimation netlist for use in some third-party synthesis tools.
fft256.html	The MegaCore function report file

block is required and the first data are available after $256 \times 2 \times 10\text{ ns} \approx 5\text{ }\mu\text{s}$, as indicated by the `source_sop` signal; see Fig. 6.22a. After 256 clock cycles all output data are transmitted as shown by the pulse in the `source_eop` signal (see Fig. 6.22b) and the FFT will output the next block. Notice that the output data shows little quantization, but have a block exponent of -3 , i.e., are scaled by $1/8$. This is a result of the block floating-point format used inside the block to minimize the quantization noise in the multistage FFT computation. To unscale use a barrelshifter and shift all real and imaginary data according to this exponent value.

6.20

The design from the previous example runs at 213.31 MHz using the TimeQuest 85C slow model and requires 4811 LEs, 18 embedded multipliers of size 9×9 bits (i.e., nine blocks of size 18×18 bits), and 20 M9Ks embedded memory blocks. If we compare this with the estimation of 11 M9Ks, 18 multipliers, 4581 LEs given by the FFT core toolbench, we observe no error for the multiplier, a 5% error for the LEs, and an 81% error for the M9Ks estimation.

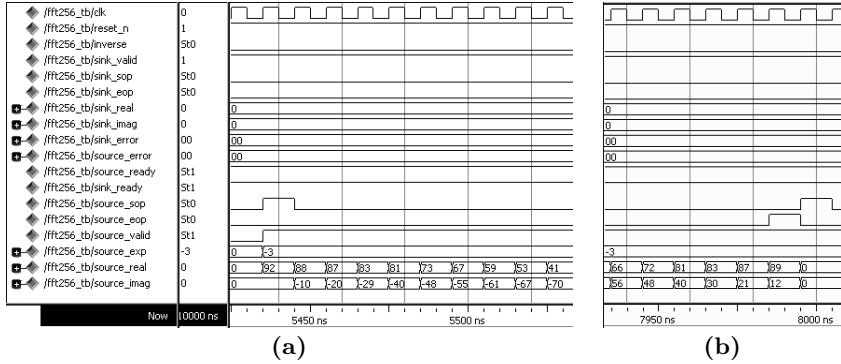


Fig. 6.22. FFT core simulation output results. (a) Start of the output frame. (b) End of the output frame

6.3 Fourier-Related Transforms

The *discrete cosine transform* (DCT) and *discrete sine transform* (DST) are not DFTs, but they can be computed using a DFT. However, DCTs and DSTs cannot be used directly to compute fast convolution, by multiplying the transformed spectra and an inverse transform, i.e., the convolution theorem does *not* hold. The applications for DCTs and DSTs are therefore not as broad as those for FFTs, but in some applications, like image compression, DCTs are (due to their close relationship to the Kahunen–Loevé transform) very popular. However, because DCTs and DSTs are defined by sine and cosine “kernels,” they have a close relation to the DFT, and will be presented in this chapter. We will begin with the definition and properties of DCTs and DSTs, and will then present an FFT-like fast computation algorithm to implement the DCT. All DCTs obey the following transform pattern observed by Wang [204]:

$$X[k] = \sum_n x[n] C_N^{n,k} \longleftrightarrow x[n] = \sum_k X[k] C_N^{n,k}. \quad (6.55)$$

The kernel functions $C_N^{n,k}$, for four different DCT instances, are defined by

$$\text{DCT-I: } C_N^{n,k} = \sqrt{2/N} c[n] c[k] \cos\left(nk \frac{\pi}{N}\right) \quad n, k = 0, 1, \dots, N$$

$$\text{DCT-II: } C_N^{n,k} = \sqrt{2/N} c[k] \cos\left(k(n + \frac{1}{2}) \frac{\pi}{N}\right) \quad n, k = 0, 1, \dots, N - 1$$

$$\text{DCT-III: } C_N^{n,k} = \sqrt{2/N} c[n] \cos\left(n(k + \frac{1}{2}) \frac{\pi}{N}\right) \quad n, k = 0, 1, \dots, N - 1$$

$$\text{DCT-IV: } C_N^{n,k} = \sqrt{2/N} \cos\left((k + \frac{1}{2})(n + \frac{1}{2}) \frac{\pi}{N}\right) \quad n, k = 0, 1, \dots, N - 1,$$

where $c[m] = 1$ except $c[0] = 1/\sqrt{2}$. The DST has the same structure, but the cosine terms are replaced by sine terms. DCTs have the following properties:

- 1) DCTs implement functions using cosine bases.
- 2) All transforms are *orthogonal*, i.e., $\mathbf{C} \times \mathbf{C}^t = k[n]\mathbf{I}$.
- 3) A DCT is a real transform, unlike the DFT.
- 4) DCT-I is its own inverse.
- 5) DCT-II is the inverse of DCT-III, and vice versa.
- 6) DCT-IV is its own inverse. Type IV is symmetric, i.e., $\mathbf{C} = \mathbf{C}^t$.
- 7) The convolution property of the DCT is not the same as the convolution multiplication relationship in the DFT.
- 8) The DCT is an approximation of the Kahanen–Loevé transformation (KLT).

The two-dimensional 8×8 transform of the DCT-II is used most often in image compression, i.e., in the H.261, H.263, and MPEG standards for video and in the JPEG standard for still images. Because the two-dimensional transform is *separable* into two dimensions, we compute the two-dimensional DCT by row transforms followed by column transforms, or vice versa (Exercise 6.17, p. 469). We will therefore focus on the implementation of one-dimensional transforms.

6.3.1 Computing the DCT Using the DFT

Narasimha and Peterson [205] have introduced a scheme describing how to compute the DCT with the help of the DFT [206, p. 50]. The mapping of the DCT to the DFT is attractive because we can then use the wide variety of FFT-type algorithms. Because DCT-II is used most often, we will further develop the relationship of the DFT and DCT-II. To simplify the representation, we will skip the scaling operation, since it can be included at the end of the DFT or FFT computation. Assuming that the transform length is even, we can rewrite the DCT-II transform

$$X[k] = \sum_{n=0}^{N-1} x[n] \cos \left(k \left(n + \frac{1}{2} \right) \frac{\pi}{N} \right), \quad (6.56)$$

using the following permutation:

$$\begin{aligned} y[n] &= x[2n] \quad \text{and} \quad y[N-n-1] = x[2n+1] \\ \text{for } n &= 0, 1, \dots, N/2 - 1. \end{aligned}$$

It follows then that:

$$\begin{aligned} X[k] &= \sum_{n=0}^{N/2-1} y[n] \cos \left(k(2n + \frac{1}{2}) \frac{\pi}{N} \right) \\ &\quad + \sum_{n=0}^{N/2-1} y[N-n-1] \cos \left(k(2n + \frac{3}{2}) \frac{\pi}{N} \right) \end{aligned}$$

$$X[k] = \sum_n y[n] \cos \left(k(2n + \frac{1}{2}) \frac{\pi}{N} \right). \quad (6.57)$$

If we now compute the DFT of $y[n]$ denoted with $Y[k]$, we find that

$$\begin{aligned} X[k] &= \Re(W_{4N}Y[k]) \\ &= \cos \left(\frac{\pi k}{2N} \right) \Re(Y[k]) - \sin \left(\frac{\pi k}{2N} \right) \Im(Y[k]). \end{aligned} \quad (6.58)$$

This can be easily transformed in a C or MATLAB program (see Exercise 6.17, p. 469), and can be used to compute the DCT with the help of a DFT or FFT.

6.3.2 Fast Direct DCT Implementation

The symmetry properties of DCTs have been used by Byeong Lee [207] to construct an FFT-like DCT algorithm. Because of its similarities to a radix-2 Cooley–Tukey FFT, the resulting algorithm is sometimes referred to as the *fast DCT* or simply FCT. Alternatively, a fast DCT algorithm can be developed using a matrix structure [208]. A DCT can be obtained by “transposing” an inverse DCT (IDCT) since the DCT is known to be an orthogonal transform. IDCT Type II was introduced in (6.55) and, noting that $\hat{X}[k] = c[k]X[k]$, it follows that:

$$x[n] = \sum_{k=0}^{N-1} \hat{X}[k]C_N^{n,k}, \quad n = 0, 1, \dots, N-1. \quad (6.59)$$

Decomposing $x[n]$ into even and odd parts it can be shown that $x[n]$ can be reconstructed by two $N/2$ DCTs, namely

$$G[k] = \hat{X}[2k], \quad (6.60)$$

$$H[k] = \hat{X}[2k+1] + \hat{X}[2k-1], \quad k = 0, 1, \dots, N/2-1. \quad (6.61)$$

In the time domain, we get

$$g[n] = \sum_{k=0}^{N/2-1} G[k]C_{N/2}^{n,k}, \quad (6.62)$$

$$h[n] = \sum_{k=0}^{N/2-1} H[k]C_{N/2}^{n,k}, \quad k = 0, 1, \dots, N/2-1. \quad (6.63)$$

The reconstruction becomes

$$x[n] = g[n] + 1/(2C_N^{n,k})h[n], \quad (6.64)$$

$$x[N-1-n] = g[n] - 1/(2C_N^{n,k})h[n], \quad (6.65)$$

$$n = 0, 1, \dots, N/2-1.$$

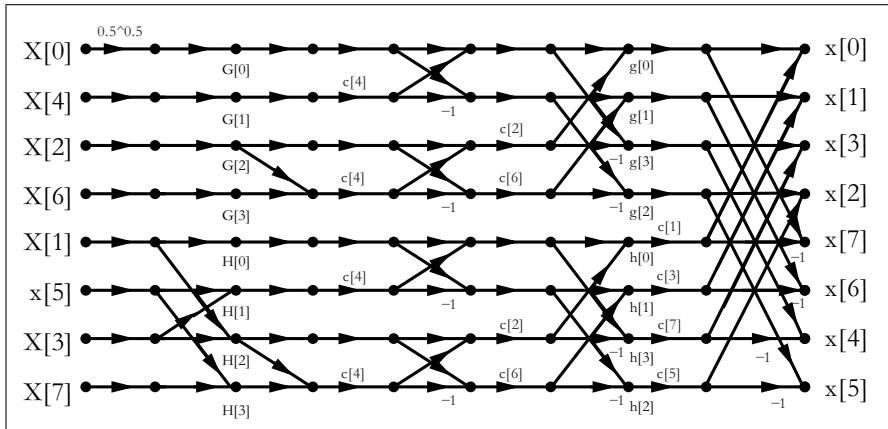


Fig. 6.23. 8-point fast DCT flow graph with the short-hand notation $c[p] = 1/(2 \cos(p\pi/16))$

By repeating this process, we can decompose the DCT further. Comparing (6.62) with the radix-2 FFT twiddle factor shown in Fig. 6.13 (p. 440) shows that a division seems to be necessary for the FCT. The twiddle factors $1/(2C_N^{n,k})$ should therefore be precomputed and stored in a table. Such a table approach is also appropriate for the Cooley–Tukey FFT, because the “online” computation of the trigonometric function is, in general, too time consuming. We will demonstrate the FCT with the following example.

Example 6.21: A 8-point FCT

For an 8-point FCT (6.60)–(6.65) become

$$G[k] = \hat{X}[2k], \quad (6.66)$$

$$H[k] = \hat{X}[2k+1] + \hat{X}[2k-1], \quad k = 0, 1, 2, 3. \quad (6.67)$$

and in the time domain we get

$$g[n] = \sum_{k=0}^3 G[k] C_4^{n,k}, \quad (6.68)$$

$$h[n] = \sum_{k=0}^3 H[k] C_4^{n,k}, \quad n = 0, 1, 2, 3. \quad (6.69)$$

The reconstruction becomes

$$x[n] = g[n] + 1/(2C_8^{n,k})h[n], \quad (6.70)$$

$$x[N-1-n] = g[n] - 1/(2C_8^{n,k})h[n], \quad n = 0, 1, 2, 3. \quad (6.71)$$

Equations (6.66) and (6.67) form the first stage in the flow graph in Fig. 6.23, and (6.70) and (6.71) build the last stage in the flow graph. 6.21

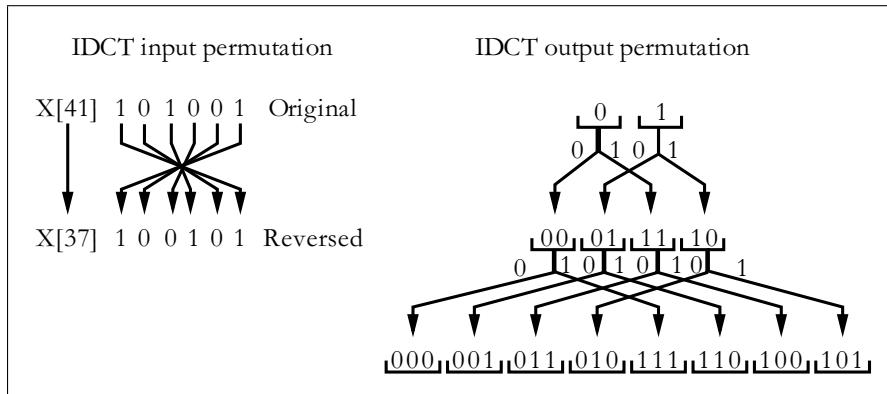


Fig. 6.24. Input and output permutation for the 8-point fast DCT

In Fig. 6.23, the input sequence $\hat{X}[k]$ is applied in bit-reversed order. The order of the output sequence $x[n]$ is generated in the following manner: starting with the set $(0, 1)$ we form the new set by adding a prefix 0 and 1. For the prefix 1, all bits of the previous pattern are inverted. For instance, from the sequence 10 we get the two babies 010 and $1\bar{1}\bar{0} = 101$. This scheme is graphically interpreted in Fig. 6.24.

Exercises

Note: If you have no prior experience with the Quartus II software, refer to the case study found in Sect. 1.4.3, p. 32. If not otherwise noted use the EP4CE115F29C7 from the Cyclone IV E family for the Quartus II synthesis evaluations.

6.1: Compute the 3-dB bandwidth, first zero, maximum sidelobe, and decrease per octave, for a rectangular and triangular window using the Fourier transform.

- 6.2:** (a) Compute the cyclic convolution of $x[n] = \{3, 1, -1\}$ and $f[n] = \{2, 1, 5\}$.
 (b) Compute the DFT matrix \mathbf{W}_3 for $N = 3$.
 (c) Compute the DFT of $x[n] = \{3, 1, -1\}$ and $f[n] = \{2, 1, 5\}$.
 (d) Now compute $Y[k] = X[k]F[k]$, followed by $\mathbf{y} = \mathbf{W}_3^{-1}\mathbf{Y}$, for the signals from part (c).

Note: use a C compiler or MATLAB for part (c) and (d).

6.3: A single spectral component $X[k]$ in the DFT computation

$$X[k] = x[0] + x[1]W_N^k + x[2]W_N^{2k} + \dots + x[N-1]W_N^{(N-1)k}$$

can be rearranged by collecting all common factors W_N^k , such that we get

$$X[k] = x[0] + W_N^k(x[1] + W_N^k(x[2] + \dots + W_N^k x[N-1] \dots)).$$

This results in a possibly recursive computation of $X[k]$. This is called the Goertzel algorithm and is graphically interpreted by Fig. 6.5 (p. 424). The Goertzel algorithm can be attractive if only a few spectral components must be computed. For the whole DFT, the effort is of order N^2 and there is no advantage compared with the direct DFT computation.

- (a) Construct the recursive signal flow graph, including input and output register, to compute a single $X[k]$ for $N = 5$.

For $N = 5$ and $k = 1$, compute all registers contents for the following input sequences:

- (b) $\{20, 40, 60, 80, 100\}$.
- (c) $\{j20, j40, j60, j80, j100\}$.
- (d) $\{20 + j20, 40 + j40, 60 + j60, 80 + j80, 100 + j100\}$.

6.4: The Bluestein chirp- z algorithm was defined in Sect. 6.1.4 (p. 424). This algorithm is graphically interpreted in Fig. 6.6 (p. 425).

- (a) Determine the CZT algorithms for $N = 4$.
- (b) Using C or MATLAB, determine the CZT for the triangular sequence $x[n] = \{0, 1, 2, 3\}$.
- (c) Using C or MATLAB, extend the length to $N = 256$, and check the CZT results with an FFT of the same length. Use a triangular input sequence, $x[n] = n$.

6.5: (a) Design a direct implementation of the nonrecursive filter for the $N = 7$ Rader algorithm.

- (b) Determine the coefficients that can be combined.

(c) Compare the realizations from (a) and (b) in terms of realization effort.

6.6: Design a length $N = 3$ Winograd DFT algorithm and draw the signal flow graph.

6.7: (a) Using the two-dimensional index transform $n = 3n_1 + 2n_2 \bmod 6$, with $N_1 = 2$ and $N_2 = 3$, determine the mapping (6.18) on p. 436. Is this mapping bijective?

(b) Using the two-dimensional index transform $n = 2n_1 + 2n_2 \bmod 6$, with $N_1 = 2$ and $N_2 = 3$, determine the mapping (6.18) on p. 436. Is this mapping bijective?

(c) For $\gcd(N_1, N_2) > 1$, Burrus [175] found the following conditions such that the mapping is bijective:

$$A = aN_2 \text{ and } B \neq bN_1 \text{ and } \gcd(a, N_1) = \gcd(B, N_2) = 1$$

or

$$A \neq aN_2 \text{ and } B = bN_1 \text{ and } \gcd(A, N_1) = \gcd(b, N_2) = 1,$$

with $a, b \in \mathbb{Z}$. Suppose $N_1 = 9$ and $N_2 = 15$. For $A = 15$, compute all possible values for $B \in \mathbb{Z}_{20}$.

6.8: For $\gcd(N_1, N_2) = 1$, Burrus [175] found that in the following conditions the mapping is bijective:

$$A = aN_2 \text{ and/or } B = bN_1 \text{ and } \gcd(A, N_1) = \gcd(B, N_2) = 1, \quad (6.72)$$

with $a, b \in \mathbb{Z}$. Assume $N_1 = 5$ and $N_2 = 8$. Determine whether the following mappings are possibly bijective index mappings:

- (a) $A = 8, B = 5$.
- (b) $A = 8, B = 10$.
- (c) $A = 24, B = 15$.
- (d) For $A = 7$, compute all valid $B \in \mathbb{Z}_{20}$.
- (e) For $A = 8$, compute all valid $B \in \mathbb{Z}_{20}$.

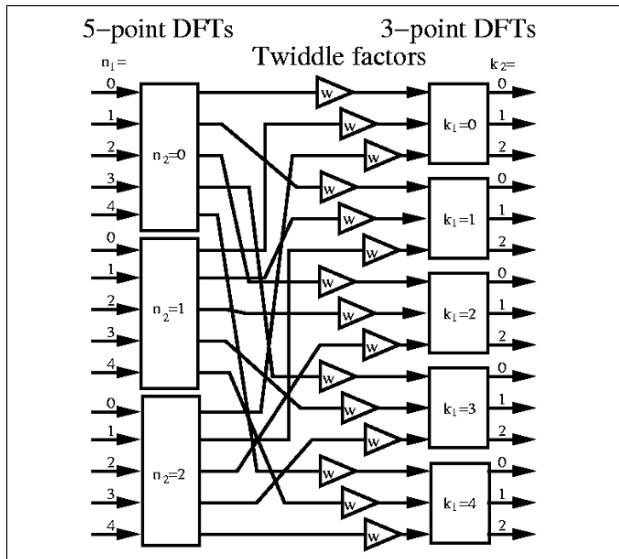


Fig. 6.25. Incomplete 16-point radix-4 FFT signal flow graph

- 6.9:** (a) Draw the signal flow graph for a radix-2 DIF algorithm where $N = 16$.
 (b) Write a C or MATLAB program for the DIF radix-2 FFT.
 (c) Test your FFT program with a triangular input $x[n] = n+jn$ with $n \in [0, N-1]$.

- 6.10:** (a) Draw the signal flow graph for a radix-2 DIT algorithm where $N = 8$.
 (b) Write a C or MATLAB program for the DIT radix-2 FFT.
 (c) Test your FFT program with a triangular input $x[n] = n+jn$ with $n \in [0, N-1]$.

- 6.11:** For a common-factor FFT the following 2D DFT (6.24; p. 438) is used:

$$X[k_1, k_2] = \sum_{n_2=0}^{N_2-1} W_N^{n_2 k_2} \left(W_N^{n_2 k_1} \sum_{n_1=0}^{N_1-1} x[n_1, n_2] W_{N_1}^{n_1 k_1} \right) \quad (6.73)$$

- (a) Compile a table for the index map for a $N = 16$ radix-4 FFT with: $n = 4n_1 + n_2$ and $k = k_1 + 4k_2$, and $0 \leq n_1, k_1 \leq N_1$ and $0 \leq n_2, k_2 \leq N_2$.
 (b) Complete the signal flow graph (\mathbf{x} , \mathbf{X} and twiddle factors) for the $N = 16$ radix 4 shown in Fig. 6.25.
 (c) Compute the 16-point FFT for $x = [0, 1, 0, 0, 0, 2, 0, 0, 0, 3, 0, 0, 0, 4, 0, 0]$ using the following steps:
 (c1) Map the input data and compute the DFTs of the first stage.
 (c2) Multiply the (none-zero DFTs) with the twiddle factors (hint: $w = \exp(-j2\pi/16)$).
 (c3) Compute the second level DFT.
 (c4) Sort the output sequence X in the right order (use two fractional digits).
 Note: Consider using a C compiler or MATLAB for part (c).

- 6.12:** Draw the signal flow graph for an $N = 12$ Good–Thomas FFT, such that no crossings occur in the signal flow graph.
 (Hint: Use a 3D representation of the row and column DFTs)

6.13: The index transform for FFTs by Burrus and Eschenbacher [209] is given by

$$n = N_2 n_1 + N_1 n_2 \bmod N \quad \begin{cases} 0 \leq n_1 \leq N_1 - 1 \\ 0 \leq n_2 \leq N_2 - 1, \end{cases} \quad (6.74)$$

and

$$k = N_2 k_1 + N_1 k_2 \bmod N \quad \begin{cases} 0 \leq k_1 \leq N_1 - 1 \\ 0 \leq k_2 \leq N_2 - 1. \end{cases} \quad (6.75)$$

- (a) Compute the mapping for n and k with $N_1 = 3$ and $N_2 = 4$.
- (b) Compute W^{nk} .
- (c) Substitute W^{nk} from (b) in the DFT matrix.
- (d) What type of FFT algorithm is this?
- (e) Can the Rader algorithm be used to compute the DFTs of length N_1 or N_2 ?

6.14: (a) Compute the DFT matrices \mathbf{W}_2 and \mathbf{W}_3 .

(b) Compute the Kronecker product $\mathbf{W}'_6 = \mathbf{W}_2 \otimes \mathbf{W}_3$.

(c) Compute the index for the vectors \mathbf{X} and \mathbf{x} , such that $\mathbf{X} = \mathbf{W}'_6 \mathbf{x}$ is a DFT of length 6.

(d) Compute the index mapping for $x[n]$ and $X[k]$, with $\mathbf{x} = \mathbf{W}_2^* \otimes \mathbf{W}_3^* \mathbf{X}$ being the IDFT.

6.15: The discrete Hartley transformation (DHT) is a transform for real signals. A length N transform is defined by

$$H[n] = \sum_{k=0}^{N-1} \text{cas}(2\pi nk/N) h[k], \quad (6.76)$$

with $\text{cas}(x) = \sin(x) + \cos(x)$. The relation with the DFT ($f[k] \xrightarrow{\text{DFT}} F[n]$) is

$$H[n] = \Re\{F[n]\} - \Im\{F[n]\} \quad (6.77)$$

$$F[n] = E[n] - jO[n] \quad (6.78)$$

$$E[n] = \frac{1}{2} (H[n] + H[-n]) \quad (6.79)$$

$$O[n] = \frac{1}{2} (H[n] - H[-n]), \quad (6.80)$$

where \Re is the real part, \Im the imaginary part, $E[n]$ the even part of $H[n]$, and $O[n]$ the odd part of $H[n]$.

- (a) Compute the equation for the inverse DHT.
- (b) Compute (using the frequency convolution of the DFT) the steps to compute a convolution with the DHT.
- (c) Show possible simplifications for the algorithms from (b), if the input sequence is even.

6.16: The DCT-II form is:

$$X[k] = c[k] \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} x[n] \cos \left(\frac{2\pi}{4N} (2n+1)k \right) \quad (6.81)$$

$$c[k] = \begin{cases} \sqrt{1/2} & k = 0 \\ 1 & \text{otherwise} \end{cases}. \quad (6.82)$$

- (a) Compute the equations for the inverse transform.

- (b) Compute the DCT matrix for $N = 4$.
 (c) Compute the transform of $x[n] = \{1, 2, 2, 1\}$ and $x[n] = \{1, 1, -1, -1\}$.
 (d) What can you say about the DCT of even or odd symmetric sequences?

6.17: The following MATLAB code can be used to compute the DCT-II transform (assuming even length $N = 2^n$), with the help of a radix-2 FFT (see Exercise 6.9):

```
function X = DCTII(x)
    N = length(x); % get length
    y = [ x(1:2:N); x(N:-2:2) ]; % re-order elements
    Y = fft(y); % Compute the FFT
    w = 2*exp(-i*(0:N-1)'*pi/(2*N))/sqrt(2*N); % get weights
    w(1) = w(1) / sqrt(2); % make it unitary
    X = real(w .* Y); % compute pointwise product
```

- (a) Compile the program with C or MatLab.
 (b) Compute the transform of $x[n] = \{1, 2, 2, 1\}$ and $x[n] = \{1, 1, -1, -1\}$.

6.18: Like the DFT, the DCT is a separable transform and, we can therefore implement a 2D DCT using 1D DCTs. The 2D $N \times N$ transform is given by

$$X[n_1, n_2] = \frac{c[n_1]c[n_2]}{4} \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} x[k, l] \cos\left(n_1(k + \frac{1}{2})\frac{\pi}{N}\right) \cos\left(n_2(l + \frac{1}{2})\frac{\pi}{N}\right), \quad (6.83)$$

where $c[0] = 1/\sqrt{2}$ and $c[m] = 1$ for $m \neq 0$.

Use the program introduced in Exercise 6.17 to compute an 8×8 DCT transform by

- (a) First row followed by column transforms.
- (b) First column followed by row transforms.
- (c) Direct implementation of (6.83).
- (d) Compare the results from (a) and (b) for the test data $x[k, l] = k + l$ with $k, l \in [0, 7]$

6.19: (a) Implement a first order system according to Exercise 6.3, to compute the Goertzel algorithm for $N = 5$ and $n = 1$, and 8-bit coefficient and input data, using Quartus II.

(b) Determine the registered performance **Fmax** using the **TimeQuest** slow 85C model and the used resources (LEs, multipliers, and M9Ks).

Simulate the design with the three input sequences:

- (c) $\{20, 40, 60, 80, 100\}$,
- (d) $\{j20, j40, j60, j80, j100\}$, and
- (e) $\{20 + j20, 40 + j40, 60 + j60, 80 + j80, 100 + j100\}$.

6.20: (a) Design a Component to compute the (real input) 4-point Winograd DFT (from Example 6.15, p. 452) using Quartus II. The input and output precision should be 8 bits and 10 bit, respectively.

(b) Determine the registered performance **Fmax** using the **TimeQuest** slow 85C model and the used resources (LEs, multipliers, and M9Ks).

Simulate the design with the three input sequences:

- (c) $\{40, 70, 100, 10\}$.
- (d) $\{0, 30, 60, 90\}$.
- (e) $\{80, 110, 20, 50\}$.

6.21: (a) Design a Component to compute the (complex input) 3-point Winograd DFT (from Example 6.15, p. 452) using Quartus II. The input and output precision should be 10 bits and 12 bits, respectively.

(b) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks).

(c) Simulate the design with the input sequences $\{180, 220, 260\}$.

6.22: (a) Using the designed 3- and 4-point Components from Exercises 6.20 and 6.21, use component instantiation to design a fully parallel 12-point Good–Thomas FFT similar to that shown in Fig. 6.18 (p. 451), using Quartus II. The input and output precision should be 8 bit and 12 bit, respectively.

(b) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks).

(c) Simulate the design with the input sequences $x[n] = 10n$ with $0 \leq n \leq 11$.

6.23: (a) Design a component `ccmulp` as in Algorithm 6.10 (p. 442), to compute the twiddle factor multiplication. Use three pipeline stages for the multiplier and one for the input subtraction $X - Y$, using Quartus II. The input and output precision should again be 8 bits.

(b) Conduct a simulation to ensure that the pipelined multiplier correctly computes $(70 + j50)(121 + j39)$.

(c) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) of the twiddle factor multiplier.

(d) Now implement the whole pipelined butterfly processor.

(e) Conduct a simulation, with the data from Example 6.11 (p. 442).

(f) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) of the whole pipelined butterfly processor.

6.24: (a) Compute the cyclic convolution of $x[n] = \{1, 2, 3, 4, 5\}$ and $f[n] = \{-1, 0, -2, 0, 4\}$.

(b) Compute the DFT matrix \mathbf{W}_5 for $N = 5$.

(c) Compute the DFT of $x[n]$ and $f[n]$.

(d) Now compute $Y[k] = X[k]F[k]$, followed by $\mathbf{y} = \mathbf{W}_5^{-1}\mathbf{Y}$, for the signals from part (c).

Note: use a C compiler or MATLAB for part (c) and (d).

6.25: For a common-factor FFT the following 2D DFT (6.24; p. 438) is used:

$$X[k_1, k_2] = \sum_{n_2=0}^{N_2-1} W_{N_2}^{n_2 k_2} \left(W_N^{n_2 k_1} \sum_{n_1=0}^{N_1-1} x[n_1, n_2] W_{N_1}^{n_1 k_1} \right). \quad (6.84)$$

(a) Compile a table for the index map for a $N = 15$, $N_1 = 5$, and $N_2 = 3$ FFT with $n = 3n_1 + n_2$ and $k = k_1 + 5k_2$.

(b) Complete the signal flow graph shown in Fig. 6.26 for the $N = 15$ transform.

(c) Compute the 15-point FFT for $x = [0, 1, 0, 0, 2, 0, 0, 3, 0, 0, 4, 0, 0, 5, 0]$ using the following steps:

(c1) Map the input data and compute the DFTs of the first stage.

(c2) Multiply the (nonzero DFTs) with the twiddle factors, i.e., $w = \exp(-j2\pi/15)$.

(c3) Compute the second-level DFT.

(c4) Sort the output sequence X into the right order (use two fractional digits).

Note: use a C compiler or MATLAB for part (c).

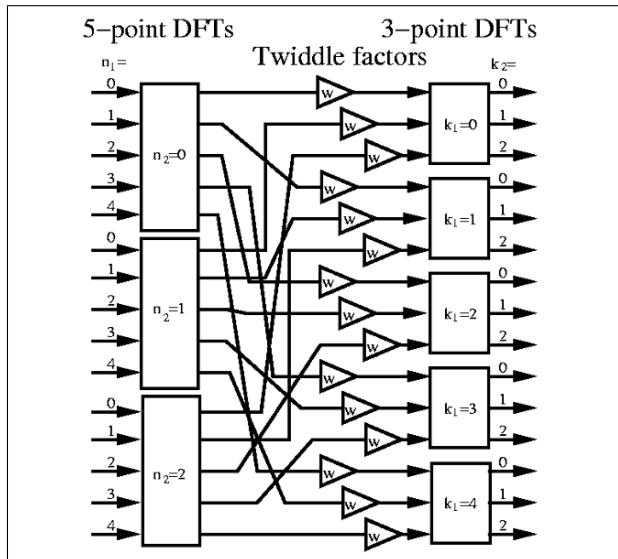


Fig. 6.26. Incomplete 15-point CFA FFT signal flow graph

6.26: For a prime-factor FFT the following 2D DFT (6.42); p. 450 is used:

$$X[k_1, k_2] = \sum_{n_2=0}^{N_2-1} W_{N_2}^{n_2 k_2} \left(\sum_{n_1=0}^{N_1-1} x[n_1, n_2] W_{N_1}^{n_1 k_1} \right). \quad (6.85)$$

- (a) Compile a table for the index map for a $N = 15, N_1 = 5, N_2 = 3$ FFT with $n = 3n_1 + 5n_2 \bmod 15$ and $k = 6k_1 + 10k_2 \bmod 15$, and $0 \leq n_1, k_1 \leq N_1$ and $0 \leq n_2, k_2 \leq N_2$.
 - (b) Draw the signal flow graph for the $N = 15$ transform.
 - (c) Compute the 15-point FFT for $x = [0, 0, 5, 0, 0, 1, 0, 0, 2, 0, 0, 3, 0, 0, 4]$ using the following steps:
 - (c1) Map the input data and compute the DFTs of the first stage.
 - (c2) Compute the second-level DFTs.
 - (c3) Sort the output sequence X into the right order (use two fractional digits).
- Note: use a C compiler or MATLAB to verify part (c).

- 6.27:** (a) Develop the table for the 5×2 Good–Thomas index map (see Theorem 6.12, p. 450) for $N_1 = 5$ and $N_2 = 2$.
- (b) Develop a program in MATLAB or C to compute the (real-input) five-point Winograd DFT using the signal flow graph shown in Fig. 6.11, p. 435. Test your code using the two input sequences {10, 30, 50, 70, 90} and {60, 80, 100, 20, 40}.
- (c) Develop a program in MATLAB or C to compute the (complex input) two-point Winograd DFT. Test your code using the two input sequences {250, 300} and {-50 + j67, -j85}.
- (d) Combine the two programs from (b) and (c) and build a Good–Thomas 5×2 FFT using the mapping from (a). Test your code using the input sequences $x[n] = 10n$ with $1 \leq n \leq 10$.

- 6.28:** (a) Design a five-point (real-input) DFT in HDL. The input and output precision should be 8 and 11 bits, respectively. Use registers for the input and output. Add a synchronous enable signal to the registers. Quantize the center coefficients using the program `csd.exe` from the CD and use a CSD coding with at least 8-bit precision.
- (b) Simulate the design with the two input sequences {10, 30, 50, 70, 90} and {60, 80, 100, 20, 40}, and match the simulation shown in Fig. 6.27.
- (c) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, embedded multipliers, and M9Ks) of the five-point DFT.

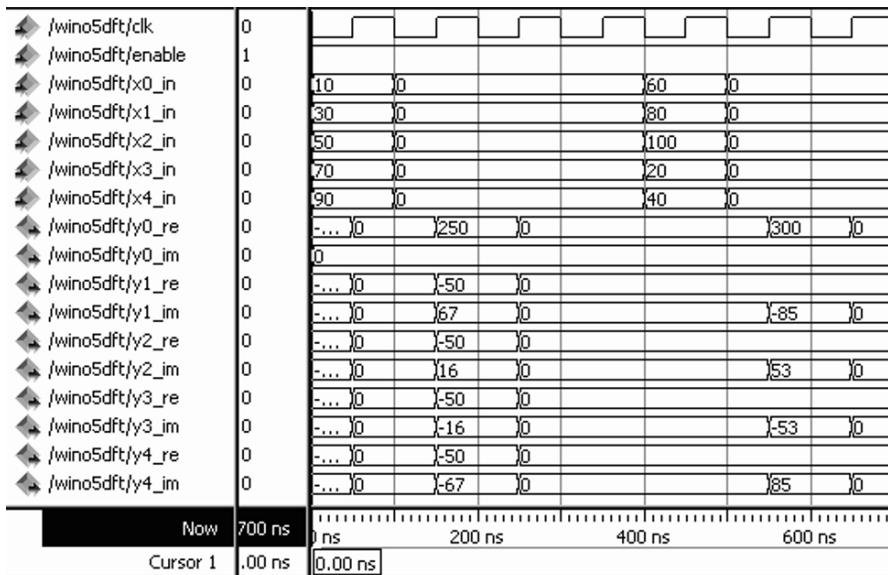


Fig. 6.27. VHDL simulation of a five-point real-input Winograd DFT

- 6.29:** (a) Design a two-point (complex input) Winograd DFT in HDL. Input and output precision should be 11 and 12 bits, respectively. Use registers for the input and output. Add a synchronous enable signal to the registers.
- (b) Simulate the design with the two input sequences {250, 300} and {-50 + j67, -j85}, and match the simulation shown in Fig. 6.28.
- (c) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, embedded multipliers, and M9Ks) of the two-point DFT.

- 6.30:** (a) Using the designed five- and two-point components from Exercises 6.28 and 6.29 use component instantiation to design a fully parallel 10-point Good–Thomas FFT similar to your software code from Exercise 6.27. The input and output precision should be 8 and 12 bits, respectively. Add an asynchronous reset for the I/O FSM and I/O registers. Use a signal ENA to indicate when a set of I/O

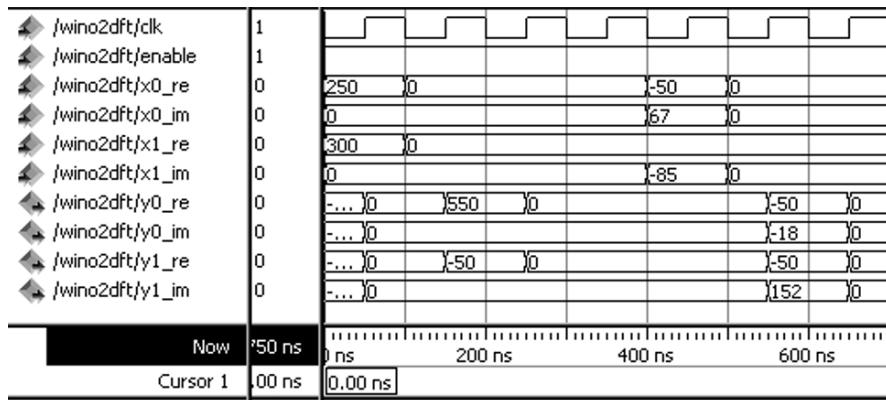
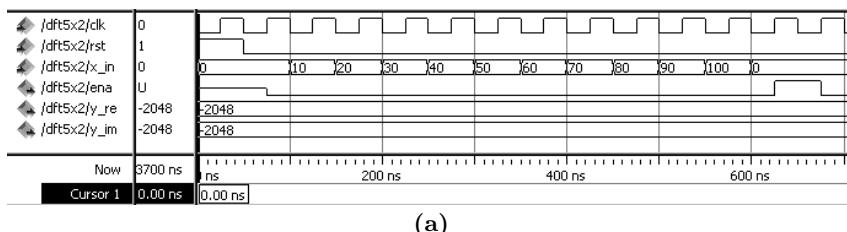


Fig. 6.28. VHDL simulation of a two-point complex-input Winograd DFT

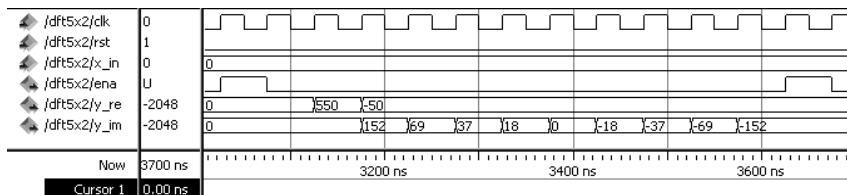
values has been transferred.

(b) Simulate the design with the input $x[n] = 10n$ with $1 \leq n \leq 10$. Try to match the simulation shown in Fig. 6.29.

(c) Determine the registered performance Fmax using the TimeQuest slow 85C model and the used resources (LEs, embedded multipliers, and M9Ks) of the 10-point Good–Thomas FFT.



(a)



(b)

Fig. 6.29. VHDL simulation of a 10-point Good–Thomas FFT. (a) Begin of frame.
(b) End of frame

6.31: Fast IDCT design.

(a) Develop a fast IDCT (MATLAB or C) code for the length-8 transform according

to Fig. 6.23 (p. 464). Note that the scaling for $X[0]$ is $\sqrt{1/2}$ and the DCT scaling $\sqrt{2/N}$ according to (6.55) is not shown in Fig. 6.23 (p. 464).

- (b) Verify your program with the MATLAB function `idct` for the sequence $X = 10, 20, \dots, 80$.
- (c) Determine the maximum bit growth for each spectral component. Hint: in MATLAB take advantage of the functions `abs`, `max`, `dctmtx`, and `sum`.
- (d) Using the program `csd.exe` from the CD determine for each coefficient $c[p] = 0.5 / \cos(p/16)$ the CSD presentation for at least 8-bit precision.
- (e) For the input sequence $X = 10, 20, \dots, 80$ compute the output in float and integer format.
- (f) Tabulate the intermediate values behind the first-, second-, and third-stage multiplication by $c[p]$. As input use the sequence X from (e) with additional four guard bits, i.e., scaled by $2^4 = 16$.

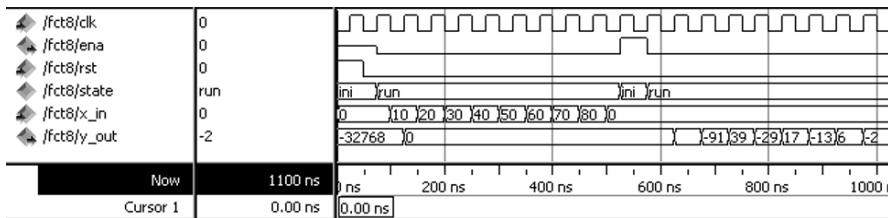


Fig. 6.30. VHDL simulation of an 8-point IDCT

- 6.32:** (a) Develop the HDL code for the length-8 transform according to Fig. 6.23 (p. 464). Include an asynchronous reset and a signal `ena` when the transform is ready. Use serial I/O. Input `x_in` should be 8 bit, as the output `y_out` and internal data format use a 14 integer format with four fractional bits, i.e., scale the input ($\times 16$) and output ($/16$) in order to implement the four fractional bits.
- (b) Use the data from Exercise 6.31(f) to debug the HDL code. Match the simulation from Fig. 6.30 for the input and output sequences.
- (c) Determine the registered performance `Fmax` using the `TimeQuest` slow 85C model and the used resources (LEs, embedded multipliers, and M9Ks) of the 8-point IDCT.
- (d) Determine the maximum output error in percent comparing the HDL and software results from Exercise 6.31.

7. Communication Systems

Several algorithms exist that enable FPGAs to outperform PDSPs by an order of magnitude, due to the fact that FPGAs can be built with bitwise implementations. Such applications are the focus of this chapter on communication systems.

For error control and cryptography, two basic building blocks are used: Galois field arithmetic and linear feedback shift registers (LFSR). Both can be efficiently implemented with FPGAs, and are discussed in Sect. 7.1. If, for instance, an N -bit LFSR is used as an M -multistep number generator, this will give an FPGA at least an MN speed advantage over a PDSPs or microprocessor.

Finally, in Sect. 7.2, receiver designed with FPGAs will demonstrate low system costs, high throughput, and the possibility of fast prototyping. A comprehensive discussion of both coherent and incoherent receivers will close this chapter.

7.1 Error Control and Cryptography

Modern communications systems, such as pagers, mobile phones or satellite transmission systems, use algorithms to correct transmission errors, since error-correction coding better utilizes the band-limited channel capacity than special modulation schemes (see Fig. 7.1). In addition, most systems also use cryptography algorithms, not just to protect messages against unauthorized listeners, but also to protect messages against unauthorized changes.

In a typical transmission scheme, such as that shown in Fig. 7.2, the *encoder* (for error correction or cryptography) is placed between the data source and the actual modulation. On the receiver side, the *decoder* is located between demodulation and the data destination (sink). Often an encoder and decoder are combined in one circuit, referred to as a CODEC.

Typical error correction and cryptographic algorithms use finite field arithmetic and are therefore more suitable for FPGAs than they are for PDSPs [211]. Bitwise operations or linear feedback shift registers (LFSR) can be very efficiently realized with FPGAs. Some CODEC schemes use large tables, and one objective when selecting the appropriate algorithms for FPGAs is therefore to find out which algorithms are most suitable. The algorithms

presented in this section are mainly based on previous publications [4] and have been used to develop a paging system for low frequencies [212–216], and an error-correction scheme for radio-controlled watches [217, 218].

It is impossible in a short section to present the whole theory of error correction and cryptography. We will present the basic ideas and suggest, for further investigation, one of the excellent textbooks in this area [190, 219–224].

7.1.1 Basic Concepts from Coding Theory

The simplest way to protect a digital transmission against random errors is to repeat the message several times. This is called *repetition code*. For a repetition of 5, for instance, the message is sent five times, i.e.,

$$0 \Leftrightarrow 00000 \quad (7.1)$$

$$1 \Leftrightarrow 11111, \quad (7.2)$$

where the left side shows the k information bits and the right side the n -bit codewords. The minimum distance between two codewords, also called the *Hamming distance* d^* , is also n and the repetition code is of the form

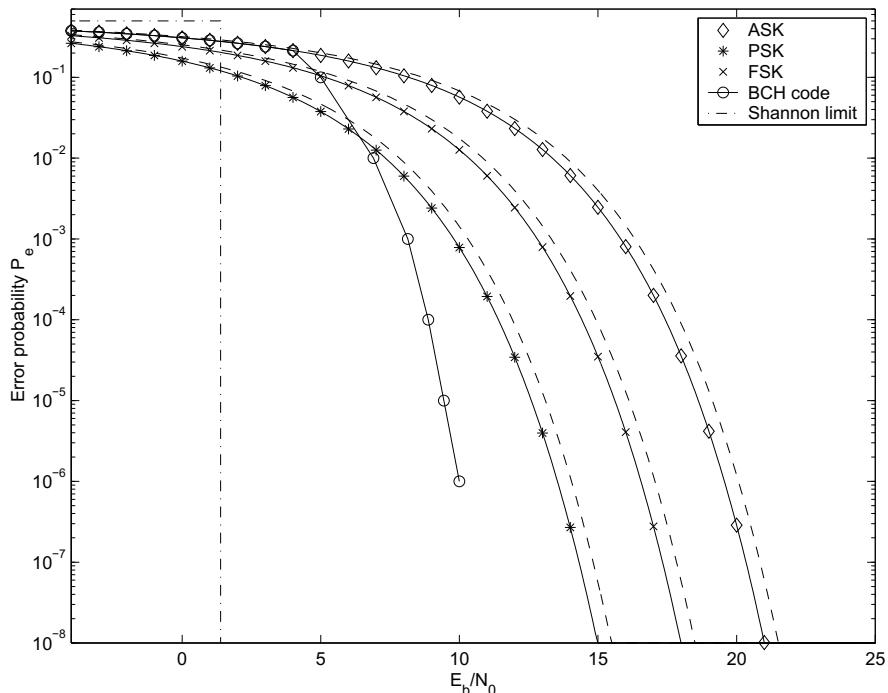


Fig. 7.1. Performance of modulation schemes [210]. **Solid line** coherent demodulation and **dashed line** incoherent demodulation

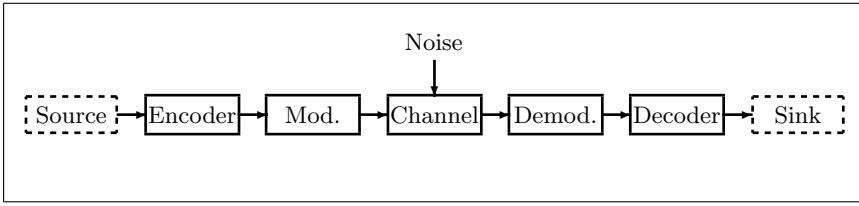


Fig. 7.2. Typical communications system configuration

$(n, k, d^*) = (5, 1, 5)$. With such a code it is possible to correct up to $\lfloor (n-1)/2 \rfloor$ random errors. But from the perspective of channel efficiency, this code is not very attractive. If our system is two-way then it is more efficient to use a technique such as a parity check and an *automatic repeat request* (ARQ) for any detected parity error. Such parity checks are used, for instance, in PC memory.

Error correction using a Hamming code. If a few more parity check bits are added, it is possible to *correct* a word with a parity error.

If the parities $P_{1,0}, P_{1,1}, P_{1,2}$, and $P_{1,3}$ are computed using modulo 2 operations, i.e., XOR, according to

$$\begin{aligned} P_{1,0} &= i_{21} \oplus i_{22} \oplus i_{23} \oplus i_{24} ; \oplus i_{25} \oplus i_{26} \oplus i_{27} \\ P_{1,1} &= i_{21} \quad \oplus i_{23} \quad \oplus i_{25} \quad \oplus i_{27} \\ P_{1,2} &= i_{21} \oplus i_{22} \quad \oplus i_{25} \oplus i_{26} \\ P_{1,3} &= i_{21} \oplus i_{22} \oplus i_{23} \oplus i_{24} \end{aligned}$$

then the parity detector is $i'_{28} (= P_{1,0})$ and three additional bits are necessary to locate the error position. Figure 7.3a shows the encoder, and Fig. 7.3b the decoder including the correction logic. On the decoder side the incoming parities are XOR'd with the newly computed parities. This forms the so-called *syndrome* ($S_{1,0} \cdots S_{1,3}$). The parities have been chosen in such a way that the syndrome pattern corresponds to the position of the bit in binary code, i.e., a $3 \rightarrow 7$ demultiplexer can be used to decode the error location.

For a more compact representation of the decoder, the following parity check matrix \mathbf{H} can be used:

$$\mathbf{H} = \left[\mathbf{P}^T : \mathbf{I} \right] = \left[\begin{array}{ccccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{array} \right]. \quad (7.3)$$

It is possible to describe the encoder using a generator matrix. $\mathbf{G} = [\mathbf{I} : \mathbf{P}]$, i.e., the generator matrix consists of a systematic identity matrix \mathbf{I} followed by the parity-bits matrix \mathbf{P} . A codeword \mathbf{v} is computed by multiplying (modulo 2) the information word \mathbf{i} with the generator matrix \mathbf{G} :

$$\mathbf{v} = \mathbf{i} \times \mathbf{G}. \quad (7.4)$$

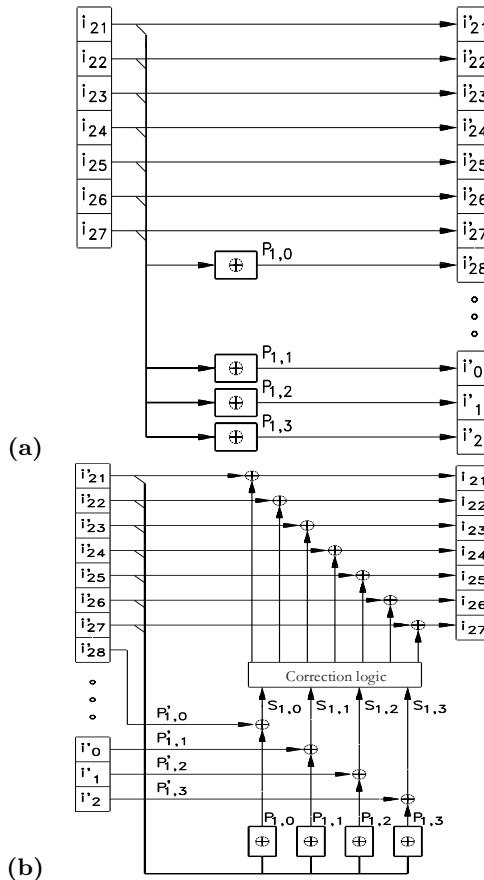


Fig. 7.3. (a) Coder and (b) Decoder for Hamming code

The (de)coders shown in Fig. 7.3 are those for a (11,7,3) Hamming code, and it is possible to detect *and* correct one error. In general, it can be shown that for 4 parity bits, up to 15 information bits, can be used, i.e., a (15,11,3) Hamming code has been *shortened* to a (11,7,3) code.

A Hamming code with distance 3 generally has a $(2^m - 1, 2^m - m, 3)$ structure. The dates in radio-controlled watches, for instance, are coded with 22 bits, and a (31,26,3) Hamming code can be shortened to a (27,22,3) code to achieve a single-error correcting code. The parity check matrix becomes:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Table 7.1. Estimated effort for error correction with Hamming code

Block Hamming code	CLB effort for		
	Minutes (11,7,3)	Hours (10,6,3)	Date (27,22,3)
Register	6	6	14
Syndrome computation	5	5	16
Correction logic	4	4	22
Output register	4	4	11
Sum	19	19	63
Total		101	

Again, the syndromes can be sorted in such a way that the correction logic is a simple $5 \rightarrow 22$ demultiplexer.

Table 7.1 shows the estimated effort in CLBs using Xilinx FPGAs for an error-correction unit for radio-controlled watches that uses three separate data blocks for minutes, hours, and date.

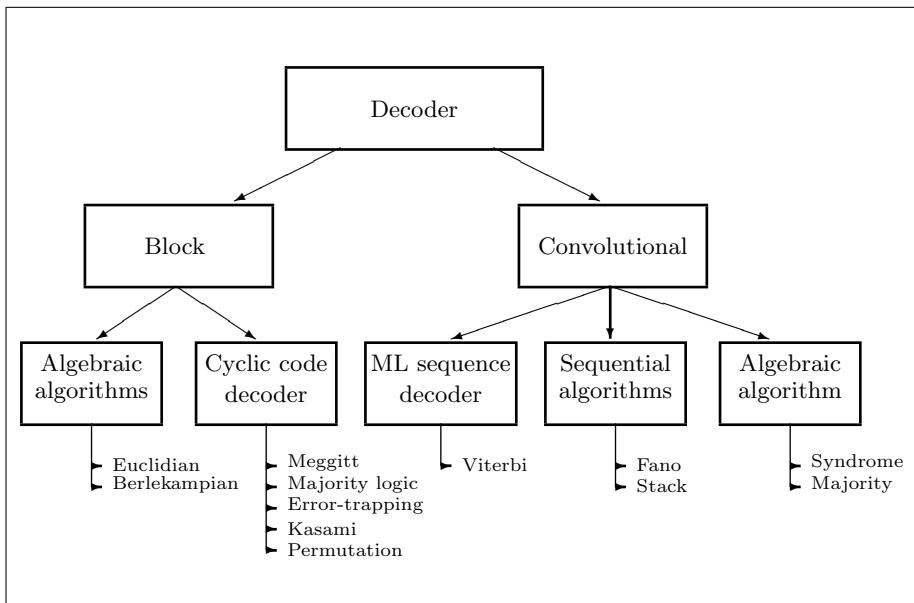
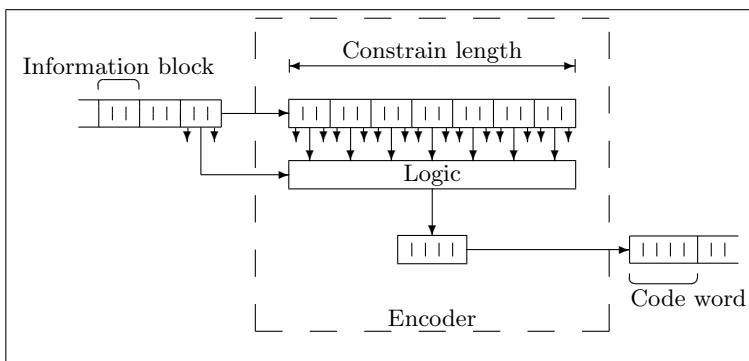
In conclusion, with an additional $3+3+5=11$ bits and the parity bits for the minutes using about 100 CLBs, it is possible to correct one error in each of the three blocks.

Survey of Error Correction Codes

After the introductory case study in the last section, commonly used codes and possible encoder and decoder implementations will be discussed next. Most often the effort for the decoder is of greater concern, since many communications systems like pager or radios use one sender and several receivers. Figure 7.4 shows a diagram of possible decoders.

Some nearly optimal decoders use huge tables and are not included in Fig. 7.4. The difference between block and convolutional codes is based on whether “memory” is used in the code generation. Both methods are characterized by the code rate R , which is the quotient of the information bits and the code length, i.e., $R = k/n$. For tree codes with memory, the actual output block, which is n bits long, depends not only on the present k information bits, but also on the previous m symbols, as shown in Fig. 7.5. Characteristics of convolution codes are the memory length $\nu = m \times k$, as well the distance profile, the free distance d_f , and the minimum distance d_m (see, for instance, [190]). Block codes can most often be constructed with algebraic methods using Galois fields, but tree codes are often only found in computer simulations.

Our discussion will be limited to *linear* codes, i.e., codes where the sum of two codewords is again a codeword, because this simplifies the decoder implementation. For linear codes, the Hamming distance can always be computed

**Fig. 7.4.** Decoder for error correction**Fig. 7.5.** Parameters of the convolutional encoders

as the difference between a codeword and the zero word, which simplifies comparisons of the performance of the code. Linear tree codes are often called *convolutional codes*, because the codes can be built using an FIR-like structure. Convolutional codes may be *catastrophic* or *noncatastrophic*. In the case of a catastrophic code, a single error will be propagated forever. It can be shown that systematic convolutional codes are always noncatastrophic. It is also common to distinguish between *random error* correction codes and *burst error* correction codes. In burst error correction, there may be a long burst of errors (or erasures). In random error correction code, the capability to cor-

rect errors is not limited to consecutive bits – the error may have a random position in the received codeword.

Coding bounds. With *coding bounds* we can compare different coding schemes. The bounds show the maximum error correction capability of the code. A decoder can never be better than the upper bound of the code, and sometimes to reduce the complexity of the decoder it is necessary to decode less than the theoretical bound.

A simple but still good, rough estimation is the Singleton bound or the Hamming bound. The *Singleton bound* states that the minimum Hamming distance d^* is upper bounded by the number of parity bits $(n - k)$. It is also known [190, p. 256] that the number of correctable errors t and the number of erasures e for a code is upper bounded by the Hamming distance. This gives the following bounds:

$$e + 2t + 1 \leq d^* \leq n - k + 1. \quad (7.5)$$

A code with $d^* = n - k + 1$ is called *maximum distance separable*, but besides the repetition code and the parity check code, there are no binary maximum distance separable codes [190, p. 431]. Following the example in the last section from Table 7.1, with 11 parity bits the upper bound can be used to correct up to five errors.

For a t -error-correcting binary code, the following *Hamming bound* provides a good estimation:

$$2^{n-k} \geq \sum_{m=0}^t \binom{n}{m}. \quad (7.6)$$

Equation (7.6) says that the possible number of parity check patterns (2^{n-k}) must be greater than or equal to the number of error patterns. If the equal sign is valid in (7.6), such codes are called *perfect codes*. A perfect code is, for instance, the Hamming code discussed in the last section. If it is desired, for instance, to find a code to protect all 44 bits transmitted in one minute for radio-controlled watches, using the maximum-available 13 parity bits, then it follows that:

$$2^{13} > \binom{44}{0} + \binom{44}{1} + \binom{44}{2} \quad \text{but} \quad (7.7)$$

$$2^{13} < \binom{44}{0} + \binom{44}{1} + \binom{44}{2} + \binom{44}{3}, \quad (7.8)$$

i.e., it should be possible to find a code with the capability to correct two random errors but none with three errors. In the following sections we will review such block encoders and decoders, and then discuss convolutional encoders and decoders.

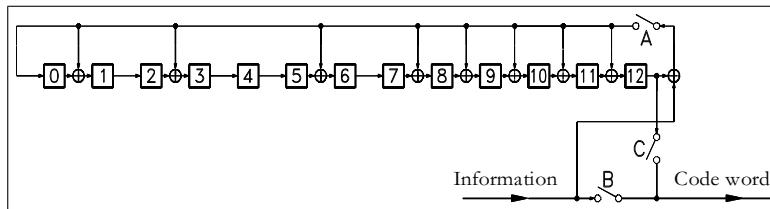


Fig. 7.6. Encoder for (57,44,6) BCH code

7.1.2 Block Codes

The linear cyclic binary BCH codes (from Bose, Chaudhuri, and Hocquenghem) and the subclass of Reed–Solomon codes, consist of a large class of block codes. BCH codes have various known efficient decoders in the time and frequency domains. In the following, we will illustrate the shortening of a (63,50,6) to a (57,44,6) BCH code. The algorithm is discussed in detail by Blahut [190, pp. 162–6].

The code is based on a transformation of $\text{GF}(2^6)$ to $\text{GF}(2)$. To describe $\text{GF}(2^6)$, a primitive polynomial of degree 6 is needed, such as $P(x) = x^6 + x + 1$. To compute the generator polynomial, the least common multiple of the first $d-1 = 5$ minimal polynomials in $\text{GF}(2^6)$ must be computed. If α denotes a primitive element in $\text{GF}(2^6)$, it follows then that $\alpha^0 = 1$ and $m_{1(x)} = x - 1$. The minimum polynomials of α , α^2 and α^4 are identical $m_{\alpha(x)} = x^6 + x + 1$, and the minimum polynomial to α^3 is $m_{\alpha^3(x)} = x^6 + x^4 + x^2 + x + 1$. It is now possible to build the generator polynomial, $g(x)$:

$$g(x) = m_{1(x)} \times m_{\alpha(x)} \times m_{\alpha^3(x)} \quad (7.9)$$

$$= x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^6 + x^3 + x + 1. \quad (7.10)$$

Using this generator polynomial (to compute the parity bits), it is now a straight forward procedure to build the encoder and decoder.

Encoder. Since a systematic code is desired, the first codeword bits are identical with the information bits. The parity bits $p(x)$ are computed by modulo reduction of the information bits $i(x)$ shifted in order to get a systematic code according to:

$$p(x) = i(x) \times x^{n-k} \bmod g(x). \quad (7.11)$$

Such a modulo reduction can be achieved with a recursive shift register as shown in Fig. 7.6. The circuit works as follows: In the beginning, switches A and B are closed and C is open. Next, the information bits are applied (MSB first) and directly transferred to the codeword. At the same time, the recursive shift register computes the parity bits. After the information bits are all processed, switches A and B are opened and C is closed. The parity bits are now shifted into the codeword.

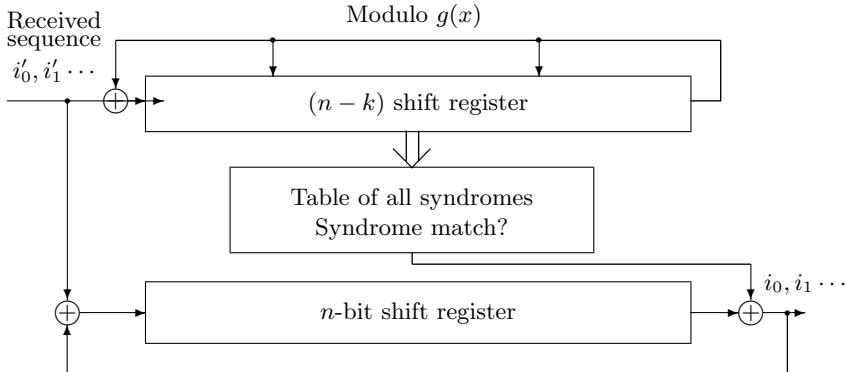


Fig. 7.7. Basic blocks of the Meggitt decoder

Decoder. The decoder is usually more complex than the encoder. A Meggitt decoder can be used for decoding in the time domain, and frequency decoding is also possible, but it needs a detailed understanding of the algebraic properties of BCH codes ([190, pp. 166–200], [219, pp. 81–107], [220, pp. 65–73]). Such frequency decoders for FPGAs are already available as intellectual property (IP) blocks, sometimes also called “virtual components,” VC (see [22, 23, 225]).

The Meggitt decoder (shown in Fig. 7.7) is very efficient for codes with only a few errors to be corrected, since the decoder uses the cyclic properties of BCH codes. Only errors in the highest bit position are corrected and then a cyclic shift is computed, so that eventually all corrupted bits pass the MSB position and are corrected.

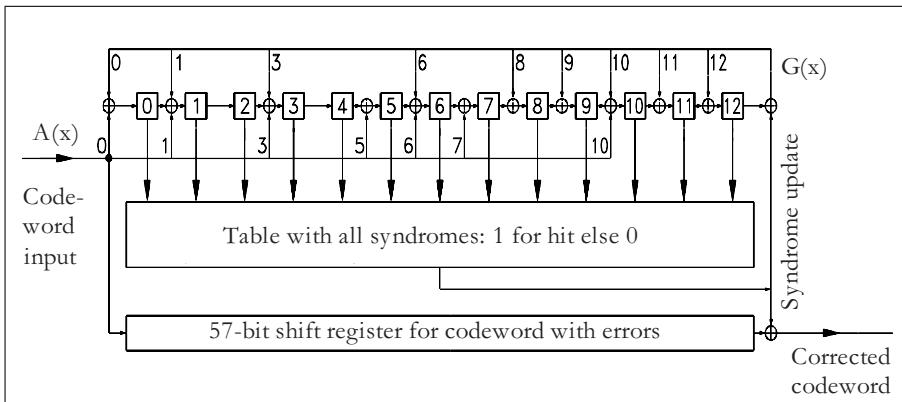
In order to use a shortened code and to regain the cyclic properties of the codes, a forward incoupling of the received data $a(x)$ must be computed. This condition can be gained for code shortened by b bits using the condition

$$s(x) = a(x)i(x) \bmod g(x) = x^{n-k+b}i(x) \bmod g(x). \quad (7.12)$$

For the shortened (57,44,6) BCH code this becomes

$$\begin{aligned} a(x) &= x^{63-50+6} \bmod g(x) = x^{19} \bmod g(x) \\ &= x^{19} \bmod (x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^6 + x^3 + x + 1) \\ &= x^{10} + x^7 + x^6 + x^5 + x^3 + x + 1. \end{aligned}$$

The developed code has the ability to correct two errors. If only the error in the MSB need be corrected, a total of $1 + \binom{56}{1} = 1 + 56 = 57$ different error patterns must be stored, as shown in Table 7.2. The 57 syndrome values can be computed through a simulation and are listed in [218, B.3].

**Fig. 7.8.** Meggitt decoder for (57,44,6) BCH code

Now all the building blocks are available for constructing the Meggitt decoder for the (57,44,6) BCH code. The decoder is shown in Fig. 7.8.

The Meggitt decoder has two stages. In the initialization phase, the syndrome is computed by processing the received bits modulo the generator polynomial $g(x)$. This takes 57 cycles. In the second phase, the actual error correction takes place. The content of the syndrome register is compared with the values of the syndrome table. If an entry is found, the table delivers a one, otherwise it delivers a zero. This hit bit is then XOR'd with the received bits in the shift register. In this way, the error is removed from the shift register. The hit bit is also wired to the syndrome register, to remove the error pattern from the syndrome register. Once again the syndrome and the shift register are clocked, and the next correction can be done. At the end, the shift register should include the corrected word, while the syndrome register should contain the all-zero word. If the syndrome is not zero, then more than two errors have occurred, and these cannot be corrected with this BCH code.

Our only concern for an FPGA implementation of the Meggitt decoder is the large number (13) of inputs for the syndrome table, because the LUTs of

Table 7.2. Table of possible error patterns

No.	Error pattern								
1	0	0	0	0	...	0	0	1	
2	0	0	0	0	...	0	1	1	
3	0	0	0	0	...	1	0	1	
:
56	0	1	0	0	...	0	0	1	
57	1	0	0	0	...	0	0	1	

FPGAs typically have 4 to 8 inputs. It is possible to use an external EPROM or one M9K embedded memory to implement a table of size $2^{13} \times 1$. The syndrome is wired to the address lines, which deliver a hit (one) for the 57 syndromes, and otherwise a zero. It is also possible to use the logic synthesis tool to compute the table with internal logic blocks on the FPGA. The Xilinx XNFOPT (used in [218]) needs 132 LUTs, each with $2^4 \times 2$ bits. If modern binary decision diagrams (BBDs) synthesizer type [226–228] are used, this number can (at the cost of additional delays) be reduced to 58 LUTs with a size of $2^4 \times 2$ bits [229]. Table 7.3 shows the estimated effort, for the Meggitt decoder using the different kinds of syndrome tables.

Table 7.3. Estimated effort for Altera FPGA devices, for the three versions of the Meggitt decoder [4]. (embedded memory is used as $2^{11} \times 1$ -ROMs)

Function group	Syndrome table		
	Using M9Ks	Only LEs	BDD [229]
Interface	36 LEs	36 LEs	36 LEs
Syndrome table	2 LEs, 1 M9K	264 LEs	116 LEs
64-bit FIFO	64 LEs	64 LEs	64 LEs
Meggitt decoder	12 LEs	12 LEs	12 LEs
State machine	21 LEs	21 LEs	21 LEs
Total	135 LEs, 1 M9K	397 LEs	249 LEs

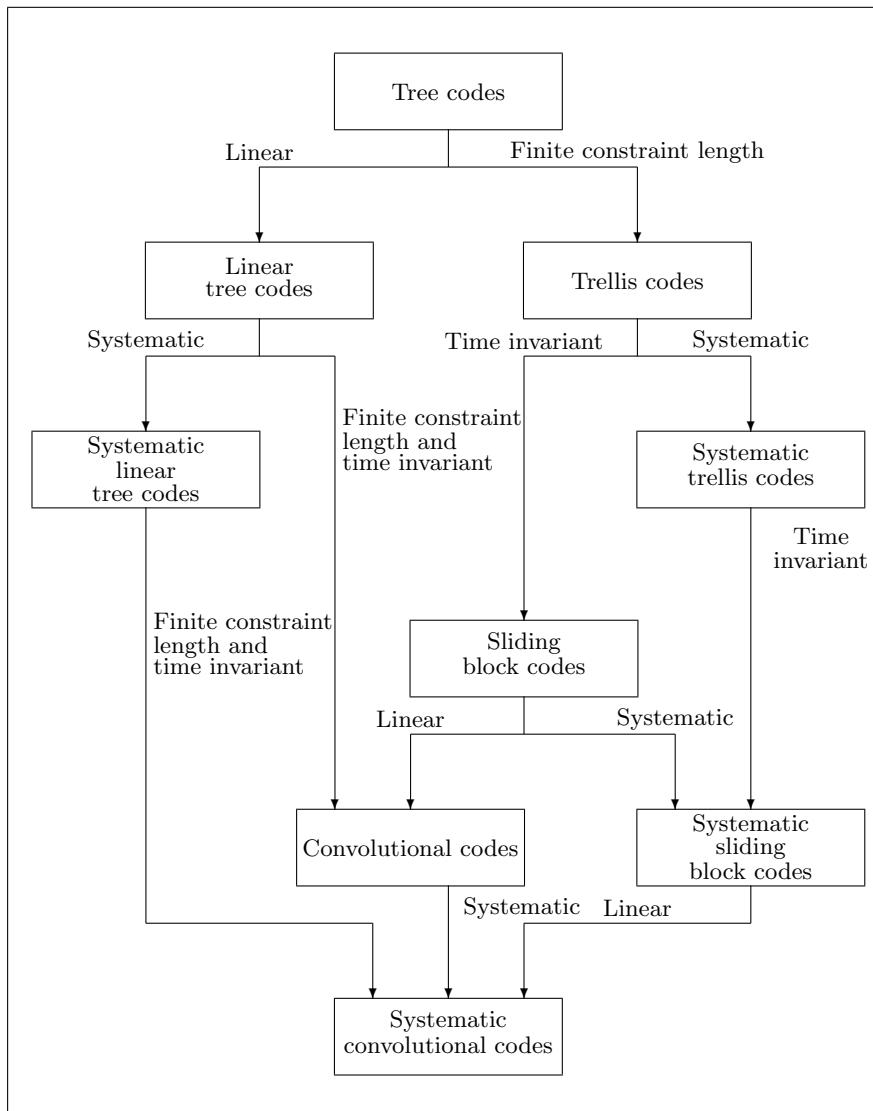
7.1.3 Convolutional Codes

We also want to explore the kind of convolutional error-correcting decoders that are suitable for an FPGA realization. To simplify the discussion, the following constraints are defined, which are typical for communications systems:

- The code should minimize the complexity of the decoder. Encoder complexity is of less concern.
- The code is linear systematic.
- The code is convolutional.
- The code should allow random error correction.

A systematic code is stipulated to allow a power-down mode, in which only incoming bits are received without error correction [217]. A random-error-correction code is stipulated if the channel is slow fading.

Figure 7.9 shows a diagram of the possible tree codes, while Fig. 7.4 (p. 480) shows possible decoders. Fano and stack decoders are not very suitable for an FPGA implementation because of the complexity of organizing a

**Fig. 7.9.** Survey of tree codes [190]

stack [216]. A conventional μ P/ μ C realization is much more suitable here. In the following sections, maximum-likelihood sequence decoders and algebraic algorithms are compared regarding hardware complexity, measured in CLBs usage for the Xilinx FPGAs, and achievable error correction.

Viterbi maximum likelihood sequence decoder. The Viterbi decoder deals with an erroneous sequence by determining the corresponding sender

sequence with the minimum Hamming distance. Put differently, the algorithm finds the optimal path through the trellis diagram, and is therefore an *optimal* memoryless noisy-sequence estimator (MLSE).

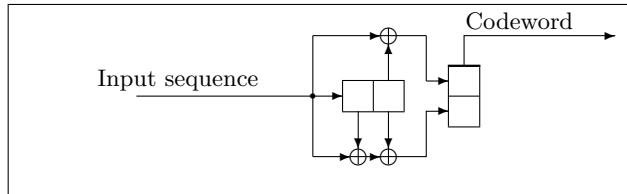


Fig. 7.10. Encoder for an $R = 1/2$ convolutional decoder

The advantage of the Viterbi decoder is its constant decoding time and MLSE optimality. The disadvantage lies in its high memory requirements and resulting limitation to codes with very short constraint length. Figures 7.10 and 7.11 show an $R = k/n = 1/2$ encoder and the attendant trellis diagram. The constraint length $\nu = m \times k$ is 2, so the trellis has $2^\nu = 4$ nodes. Each node has $2^k = 2$ outgoing and at most $2^k = 2$ incoming edges. For a binary trellis ($k = 1$) like this, it is convenient to show a zero as an upward edge and a one as a downward edge.

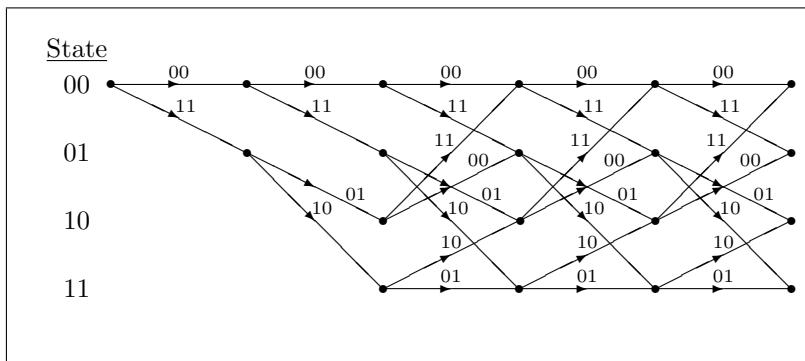


Fig. 7.11. Trellis for $R = 1/2$ convolutional decoder

For MLSE decoding it is sufficient to store only the 2^ν paths (and their metrics) passing through the nodes at a given level, because the MLSE path must pass through one of these nodes. Incoming paths with a smaller metric than the “survivor” with the highest metric need not be stored, because these paths will never be part of the MLSE path. Nevertheless, the maximum metric at any given time may not be part of the MLSE path if it is part of a short erroneous sequence. Voting down such a local error is analogous

to demodulating a digital FM signal with memory [230]. Simulation results in [190, p. 381] and [219, pp. 120–3] show that it is sufficient to construct a path memory of four to five times the constraint length. Infinite path memory yields no significant improvement.

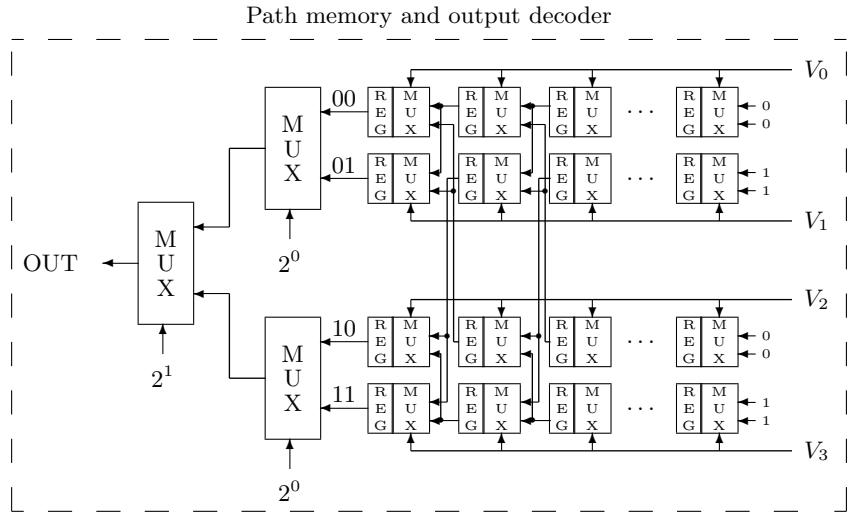


Fig. 7.12. Viterbi decoder with constraint length 4ν and $2^{\nu}=2$ nodes: path memory and output decoder

The Viterbi decoder hardware consists of three main parts: path memory with output decoder (see Fig. 7.12), survivor computation, and maximum detection (see Fig. 7.13). The path memory is $4\nu 2^\nu$ bits, consuming $2\nu 2^\nu$ CLBs. The output decoder uses $(1 + 2 + \dots + 2^{\nu-1})$ 2-to-1 multiplexers. Metric update adders, registers and comparisons are each $(\lceil \log_2(\nu * n) \rceil + 1)$ bits wide. For the maximum computation, additional comparisons, 2-to-1 multiplexers and a decoder are necessary.

The hardware for decoders with $k > 1$ seems too complex to implement with today's FPGAs. For $n > 2$ the information rate $R = 1/n$ is too low, so the most suitable code rate is $R = 1/2$. Table 7.4 lists the complexity in CLBs in a Xilinx FPGA for constraint lengths $\nu = 2, 3, 4$, and the general case, for $R = 1/2$. It can be seen that complexity increases exponentially with constraint length ν , which should thus be as short as possible. Although very few errors can be corrected in the short window allowed by such a small constraint length, the MLSE algorithm guarantees acceptable performance.

Next it is necessary to choose an appropriate generating polynomial. It is shown in the literature ([231, pp. 306–8], [232, p. 465], [221, pp. 402–7], [190, p. 367]) that, for a given constraint length, *nonsystematic* codes have better performance than systematic codes, but using a nonsystematic

Table 7.4. Hardware complexity in CLBs for an $R = 1/2$ Viterbi decoder for $\nu = 2, 3, 4$, and the general case

Function	$\nu = 2$	$\nu = 3$	$\nu = 4$	$\nu \in \mathbb{N}$
Path memory	16	48	128	$4 \times \nu \times 2^{\nu-1}$
Output decoder	1,5	3,5	6,5	$1 + 2 + \dots + 2^{\nu-2}$
Metric ΔM	4	4	4	4
Metric clear	1	2	4	$\lceil (2 + 4 + \dots + 2^{\nu-1})/4 \rceil$
Metric adder	24	64	128	$(\lceil \log_2(n\nu) \rceil + 1) \times 2^{\nu+1}$
Survivor-MUX	6	24	48	$(\lceil \log_2(n\nu) \rceil + 1) \times 2^{\nu-1}$
Metric compare	6	24	48	$(\lceil \log_2(n\nu) \rceil + 1) \times 2^{\nu-1}$
Maximum compare	4,5	14	30	$(\lceil \log_2(n\nu) \rceil + 1) \times \frac{1}{2} \times (1 + 2 + \dots + 2^{\nu-1})$
MUX	3	12	28	$\times \frac{1}{2} \times (\lceil \log_2(n\nu) \rceil + 1)$
Decoder	1	2	4	$\lceil (2 + \dots + 2^{\nu-1})/4 \rceil$
State machine	4	4	4	
Sum:	67	197.5	428.5	

code contradicts the demand for using the information bits without error correction. *Quick look in* (QLI) codes are nonsystematic convolution codes with $R = 1/2$, providing *free distance* values as good as any known code for constraint lengths $\nu = 2$ to 4 [233]. The advantage of QLI codes is that only one XOR gate is necessary for the reconstruction of the information sequence. QLIs with $\nu = 2, 3$, and 4 have a free distance of $d_f = 5, 6$, and 7, respectively [232, p. 465]. This seems to be a good compromise for low power consumption. The upper part of Table 7.5 shows the generating polynomials in octal notation.

Error-correction performance of the QLI decoder. To compute the error-correction performance of the QLI decoder, it is convenient to use the “union bound” method. Because QLI codes are linear, error sequences can be computed as a difference from the zero sequence. An MLSE decoder will make an incorrect decision if a sequence that starts at the null state, and differs from the null-word at j separate time steps, contains at least $j/2$ ones. The probability of this occurrence is

$$P_j = \begin{cases} \sum_{i=(j+1)/2}^j \binom{j}{i} p^i q^{j-i} & \text{for odd } j \\ \frac{1}{2} \binom{j}{j/2} p^{j/2} q^{j/2} + \sum_{i=j/2+1}^j \binom{j}{i} p^i q^{j-i} & \text{for even } j. \end{cases} \quad (7.13)$$

Now the only thing necessary for a bit-error probability formula is to compute the number w_j of paths with weight j for the code, which is an

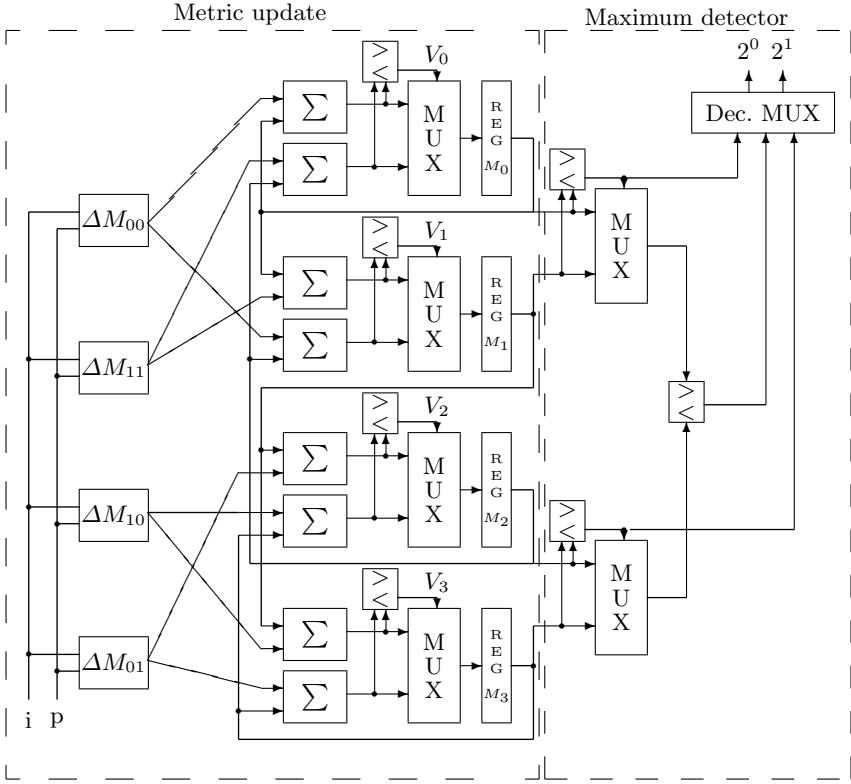


Fig. 7.13. Viterbi decoder with constraint length 4ν and $2^{\nu}=2$ nodes: metric calculation

easily programmable task [216, C.4]. Because P_j decreases exponentially with increasing j , only the first few w_j must be computed. Table 7.5 shows the w_j for $j = 0$ to 20. The total error probability can now be computed with:

$$P_b < \frac{1}{k} \sum_{j=0}^{\infty} w_j P_j. \quad (7.14)$$

Syndrome algebraic decoder. The syndrome decoder (Fig. 7.14) and encoder (Fig. 7.15), like standard block decoders, computes a number of parity bits from the data sequence. The decoder's newly computed parity bits are XOR'd with the received parity bits to create the “syndrome” word, which will be nonzero if an error occurs in transmission. The error position and value are determined from the syndrome value. In contrast to block codes, where only one generator polynomial is used, convolutional codes at data rate $R = k/n$ have $k+1$ generating polynomials. The complete generator may be written in a compact $n \times k$ generator matrix. For the encoder of Fig. 7.15 the matrix is

Table 7.5. Union-bound weights for a Viterbi decoder with $\nu = 2$ to 4 using QLI codes

Code	O1 = 7 O2 = 5	O1 = 74 O2 = 54	O1 = 66 O2 = 46
Constraint length	$\nu = 2$	$\nu = 3$	$\nu = 4$
Distance	Weight w_j		
0-4	0	0	0
5	1	0	0
6	4	2	0
7	12	7	4
8	32	18	12
9	80	49	26
10	192	130	74
11	448	333	205
12	1024	836	530
13	2304	2069	1369
14	5120	5060	3476
15	11 264	12 255	8470
16	24 576	29 444	19 772
17	53 079	64 183	43 062
18	109 396	126 260	83 346
19	103 665	223 980	147 474
20	262 144	351 956	244 458

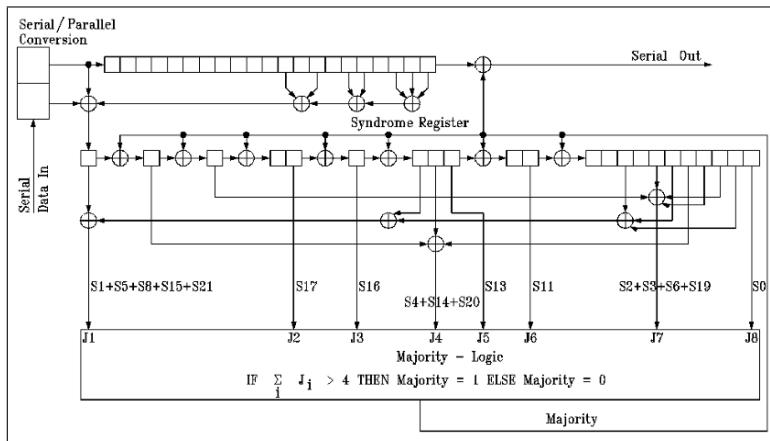


Fig. 7.14. Trial and error majority decoder with $J = 8$

$$\mathbf{G}(x) = [1 \quad x^{21} + x^{20} + x^{19} + x^{17} + x^{16} + x^{13} + x^{11} + 1]. \quad (7.15)$$

For a systematic code the matrix has the form $\mathbf{G}(x) = [\mathbf{I} : \mathbf{P}(x)]$. The parity check matrix $\mathbf{H}(x) = [-\mathbf{P}(x)^T : \mathbf{I}]$ is easily computed, given that $\mathbf{G} \times \mathbf{H}^T = \mathbf{0}$. The desired syndrome vector is thus $\mathbf{S} = \mathbf{v} \times \mathbf{H}^T$, where \mathbf{v} is the received bit sequence.

The syndrome decoder now looks up the calculated syndrome in a table to find the correct sequence. To keep the table small, only sequences with an error at the first bit position are included. If the decoder needs to correct errors of more than one bit, we cannot clear the syndrome after the correction. Instead, the syndrome value must be subtracted from a syndrome register (see the “Majority” signal in Fig. 7.14).

A 22-bit table would be necessary for the standard convolutional decoder, but it is unfortunately difficult to implement a good FPGA look-up table with more than 4 to 11 bit addresses [217]. Majority codes, a special class of syndrome-decodable codes, offer an advantage here. This type of canonical self-orthogonal code (CSOC) has exclusively ones in the first row of the $\{A_k\}$ parity check matrix (where the J columns are used as an orthogonal set to compute the syndrome) [221, p. 284]. Thus, every error in the first-bit position will cause at least $\lceil J/2 \rceil$ ones in the syndrome register. The decoding rule is therefore

$$e_0^i = \begin{cases} 1 & \text{for } \sum_{k=1}^J A_k > \lceil J/2 \rceil \\ 0 & \text{otherwise} \end{cases}. \quad (7.16)$$

Thus the name “majority code”: instead of the expensive syndrome table only a majority vote is needed. Massey [221, p. 289] has designed a class of majority codes, called trial and error codes, which, instead of evaluating the syndrome vector directly, manipulate a combination of syndrome bits to get a vector orthogonal to e_0^i . This small additional hardware cost results in slightly better error correction performance than the conventional CSOC codes. Table 7.6 lists some trial and error codes with data rate $R = 1/2$. Figure 7.14 shows a trial and error decoder with $J = 8$. Table 7.7 shows the complexity in CLBs of decoders with $J = 4$ to 10.

Error-correction capability of the trial and error decoder. To calculate the error-correction performance of trial and error codes, we must first

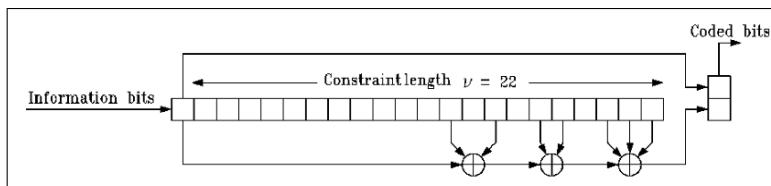


Fig. 7.15. Systematic (44, 22) encoder with rate $R = 1/2$ and constraint length $\nu = 22$

Table 7.6. Some majority-decodable “trial and error” codes [221, p. 406]

J	t_{MD}	ν	Generating polynomial	Orthogonal equation
2	1	2	$1 + x$	s_0, s_1
4	2	6	$1 + x^3 + x^4 + x^5$	$s_0, s_3, s_4, s_1 + s_5$
6	3	12	$1 + x^6 + x^7 + x^9 + x^{10} + x^{11}$	$s_0, s_6, s_7, s_9, s_1 + s_3 + s_{10},$ $s_4 + s_8 + s_{11}$
8	4	22	$1 + x^{11} + x^{13} + x^{16} + x^{17} + x^{19} + x^{20} + x^{21}$	$s_0, s_{11}, s_{13}, s_{16}, s_{17}, s_2 + s_3 + s_6 +$ $s_{19}, s_4 + s_{14} + s_{20}, s_1 + s_5 + s_8 +$ $s_{15} + s_{21}$
10	5	36	$1 + x^{18} + x^{19} + x^{27} + x^{28} + x^{29} + x^{30} + x^{32} + x^{33} + x^{35}$	$s_0, s_{18}, s_{19}, s_{27}, s_1 + s_9 + s_{28}, s_{10} +$ $s_{20} + s_{29}, s_{11} + s_{30} + s_{31},$ $s_{13} + s_{21} + s_{23} + s_{32}, s_{14} +$ $s_{33} + s_{34}, s_2 + s_3 + s_{16} + s_{24} +$ $s_{26} + s_{35}$

note that in a window twice the constraint length, the codes allow up to $\lfloor J/2 \rfloor$ -bit errors [190, p. 440]:

$$P(J) = \sum_{k=0}^{\lfloor J/2 \rfloor} \binom{2\nu}{k} p^k (1-p)^{2\nu-k}. \quad (7.17)$$

A computer simulation of 10^6 bits, in Fig. 7.16, reveals good agreement with this equation. The equivalent single-error probability P_B of an (n, k) code can be computed with

$$P(J) = P(0) = (1 - P_B)^k \quad (7.18)$$

$$\rightarrow P_B = 1 - e^{\ln(P(J))/k}. \quad (7.19)$$

Final comparison. Figure 7.16 shows the error-correction performance of Viterbi and majority decoders. For a comparable hardware cost (Viterbi,

Table 7.7. Complexity in CLBs of a majority decoder with $J = 4$ to 10

Function	$J = 4$	$J = 6$	$J = 8$	$J = 10$
Register	6	12	22	36
XOR-Gate	2	4	7	11
Majority-circuit	1	5	7	15
Sum	9	22	36	62

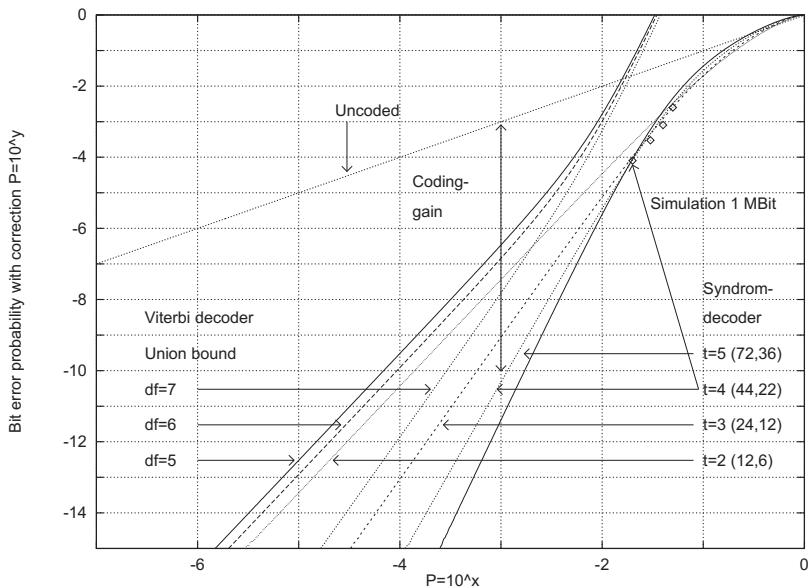


Fig. 7.16. Performance comparison of Viterbi and majority decoders

$\nu = 2$, $d_f = 5$, 67 CLBs and trial and error, $t = 5$, 62 CLBs) the better performance of the majority decoder, due to the greater constraint length permitted, is immediately apparent. The optimal MLSE property of the Viterbi algorithm cannot compensate for its short constraint length.

7.1.4 Cryptography Algorithms for FPGAs

Many communication systems use data-stream ciphers to protect relevant information, as shown in Fig. 7.17. The key sequence K is more or less a “pseudorandom sequence” (known to the sender and the receiver), and with the modulo 2 property of the XOR function, the plaintext P can be reconstructed at the receiver side, because

$$P \oplus K \oplus K = P \oplus 0 = P. \quad (7.20)$$

In the following, we compare an algorithm based on a linear-feedback shift register (LFSR) and a “data encryption standard” (DES) cryptographic algorithm. Neither algorithm requires large tables and both are suitable for an FPGA implementation.

Linear Feedback Shift Registers Algorithm

LFSRs with maximal sequence length are a good approach for an ideal security key, because they have good statistical properties (see, for instance,

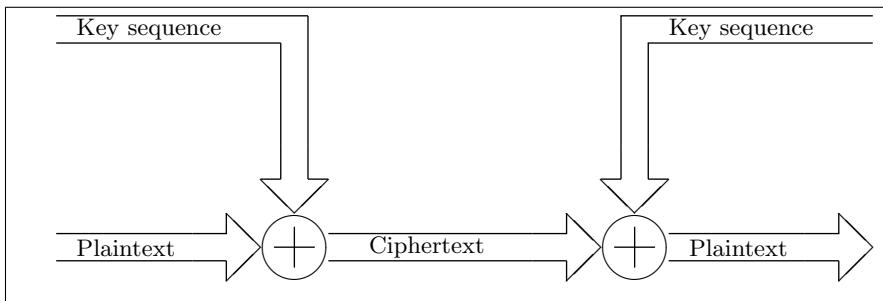


Fig. 7.17. The principle of a synchronous data-stream cipher

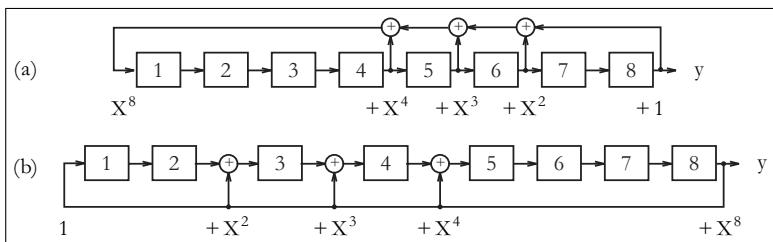


Fig. 7.18. Possible realizations of LFSRs. (a) Fibonacci configuration. (b) Galois configuration

[234, 235]). In other words, it is difficult to analyze the sequence in a cryptographic attack, an analysis called *cryptoanalysis*. Because bitwise designs are possible with FPGAs, such LFSRs are more efficiently realized with FPGAs than PDSPs. Two possible realizations of a LFSR of length 8 are shown in Fig. 7.18.

For the XOR LFSR there is always the possibility of the all-zero word, which should never be reached. If the cycle starts with any nonzero word, the cycle length is always $2^l - 1$. Sometimes, if the FPGA wakes up with an all-zero state, it is more convenient to use a “mirrored” or inverted LFSR circuit. If the all-zero word is a valid pattern and produces exactly the inverse sequence, it is necessary to substitute the XOR with a “not XOR” or XNOR gate. Such LFSRs can easily be designed using a PROCESS statement in VHDL, as the following example shows.

Example 7.1: Length 6 LFSR

The following VHDL code² implements a LFSR of length 6:

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
-----
```

² The equivalent Verilog code `lfsr.v` for this example can be found in Appendix A on page 858. Synthesis results are shown in Appendix B on page 881.

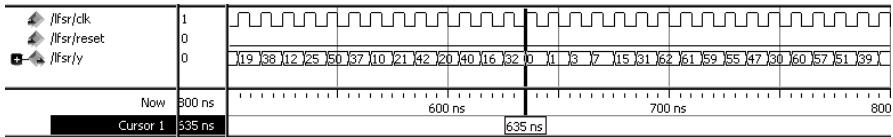


Fig. 7.19. LFSR simulation

```

ENTITY lfsr IS
    PORT ( clk      : IN STD_LOGIC;      -- System clock
           reset    : IN STD_LOGIC;      -- Asynchronous reset
           y        : OUT STD_LOGIC_VECTOR(6 DOWNTO 1));
END lfsr;
----- System output

ARCHITECTURE fpga OF lfsr IS

    SIGNAL ff  : STD_LOGIC_VECTOR(6 DOWNTO 1);

BEGIN

    PROCESS(clk, reset)
    BEGIN
        IF reset = '1' THEN          -- Implement length 6 LFSR with xnor
            ff <= (OTHERS => '0');  -- Asynchronous clear
        ELSIF rising_edge(clk) THEN
            ff(1) <= NOT (ff(5) XOR ff(6));
            FOR I IN 6 DOWNTO 2 LOOP  -- Tapped delay line:
                ff(I) <= ff(I-1);    -- shift one
            END LOOP;
        END IF;
    END PROCESS ;

    y <= ff; -- Connect to I/O cell

END fpga;

```

From the simulation of the design in Fig. 7.19, it can be concluded that the LFSR goes through all possible bit patterns, which results in the maximum sequence length of $2^6 - 1 = 63 \approx 630 \text{ ns}/10 \text{ ns}$. The design uses 6 LEs, no embedded multiplier, and has a registered performance of $\text{Fmax}=944.29 \text{ MHz}$ using the TimeQuest slow 85C model.

7.1

Note that a complete cycle of an LFSR sequence fulfills the three criteria for optimal length $2^l - 1$ pseudorandom sequences defined by Golomb [236, p. 188]:

- 1) The number of 1s and 0s in a cycle differs by no more than one.
- 2) Runs of length k (e.g., $111\cdots$ sequence, $000\cdots$ sequence) have a total fractional part of all runs of $1/2^k$.
- 3) The autocorrelation function $C(\tau)$ is constant for $\tau \in [1, n - 1]$.

LFSRs are usually constructed from primitive polynomials in GF(2) using the circuits shown in Fig. 7.18. Stahnke [222] has compiled a list of such primitive polynomials up to order 168. This paper is available online at <http://www.jstor.org>. With today's available algebraic software packages like MAPLE, MUPAD, or MAGMA such a list can easily be extended. The following is a code example for MAPLE to compute the primitive polynomials of type $x^l + x^a + 1$ with the smallest a .

```
with(numtheory): for l from 2 to 45 do
    for a from 1 by 1 to l-1 do
        if (Primitive(x^l+x^a+1) mod 2) then
            print(l,a);
            break;
        fi;
    od;
od;
```

Table 7.8 shows the necessary XOR list of the first 45 maximum length LFSRs according to Fig. 7.18a. For instance, the entry for polynomial fourteen (14, 13, 11, 9) means the primitive polynomial is

$$\begin{aligned} p_{14}(x) &= x^{14} + x^{14-13} + x^{14-11} + x^{14-9} + 1 \\ &= x^{14} + x^5 + x^3 + x + 1. \end{aligned}$$

For $l > 2$ these primitive polynomials always have “twins,” which are also primitive polynomials [237]. These are the “time” reversed versions $x^l + x^{l-a} + 1$.

Stahnke [22, XAPP52] has computed primitive polynomials of type $x^l + x^a + 1$. There are no primitive polynomials with four elements, i.e. $(x^l + x^b + x^a + 1)$ for $l < 45$. But it is possible to find polynomials of the type $x^l + x^{a+b} + x^b + x^a + 1$, which Stahnke used for those l where a polynomial of the type $x^l + x^a + 1$ ($l = 8, 12, 13$, etc.) does not exist.

The LFSRs with four elements in Table 7.8 were computed to have the maximum sum (i.e., $a + b$) for the tap exponents. We will see later that, for multistep LFSR implementations, this usually gives the minimum complexity.

If n random bits are used at once, it is possible to clock our LFSR n times. In general, it is *not* a good idea to use just the lowest n bits of our LFSR, since this will lead to weak random properties, i.e., low cryptographic security. But it is possible to compute the equation for n -bit shifts, so that only one clock cycle is needed to generate n new random bits. The necessary equation can be computed more easily if a “state-space” description of the LFSR is used, as the following example shows.

Example 7.2: Three Steps-at-Once LFSR

Let us assume a primitive polynomial of length 6, e.g., $p = x^6 + x + 1$, is used to compute random sequences. The task now is to compute three “new” bits in one clock cycle. To obtain the required equation, the state-space description of our LFSR must first be computed, i.e., $\mathbf{x}(t+1) = \mathbf{A}\mathbf{x}(t)$

Table 7.8. A list of the first 45 LFSR

l	Exponents	l	Exponents	l	Exponents
1	1	16	16, 14, 13, 11	31	31, 28
2	2, 1	17	17, 14	32	32, 30, 29, 23
3	3, 2	18	18, 11	33	33, 20
4	4, 3	19	19, 18, 17, 14	34	34, 31, 30, 26
5	5, 3	20	20, 17	35	35, 33
6	6, 5	21	21, 19	36	36, 25
7	7, 6	22	22, 21	37	37, 36, 33, 31
8	8, 6, 5, 4	23	23, 18	38	37, 36, 33, 31
9	9, 5	24	24, 23, 21, 20	39	39, 35
10	10, 7	25	25, 22	40	40, 37, 36, 35
11	11, 9	26	26, 25, 24, 30	41	41, 38
12	12, 11, 8, 6	27	27, 26, 25, 22	42	42, 39, 38, 35
13	13, 12, 10, 9	28	28, 25	43	43, 41, 40, 36
14	14, 13, 11, 9	29	29, 27	44	44, 42, 41, 37
15	15, 14	30	30, 29, 26, 24	45	45, 44, 43, 41

$$\begin{bmatrix} x_6(t+1) \\ x_5(t+1) \\ x_4(t+1) \\ x_3(t+1) \\ x_2(t+1) \\ x_1(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_6(t) \\ x_5(t) \\ x_4(t) \\ x_3(t) \\ x_2(t) \\ x_1(t) \end{bmatrix}. \quad (7.21)$$

With this state-space description, the actual values $\mathbf{x}(t)$ and the transition matrix \mathbf{A} are used to compute the new values $\mathbf{x}(t+1)$. To compute the values for $\mathbf{x}(t+2)$, simply compute $\mathbf{x}(t+2) = \mathbf{A}\mathbf{x}(t+1) = \mathbf{A}^2\mathbf{x}(t)$. The next iteration gives $\mathbf{x}(t+3) = \mathbf{A}^3\mathbf{x}(t)$. The equations for an n -step-at-once LFSR can therefore be computed by evaluating $\mathbf{A}^n \bmod 2$. For $n = 3$ it follows that:

$$\mathbf{A}^3 \bmod 2 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}. \quad (7.22)$$

As expected, for the register x_6 to x_4 there is a shift of three positions, while the other three values x_1 to x_3 are computed using an EXOR operation. The following VHDL code³ implements this three-step LFSR:

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
-----
ENTITY lfsr6s3 IS
    PORT ( clk      : IN STD_LOGIC;      -- System clock
           reset   : IN STD_LOGIC;      -- Asynchronous reset

```

³ The equivalent Verilog code `lfsr6s3.v` for this example can be found in Appendix A on page 858. Synthesis results are shown in Appendix B on page 881.

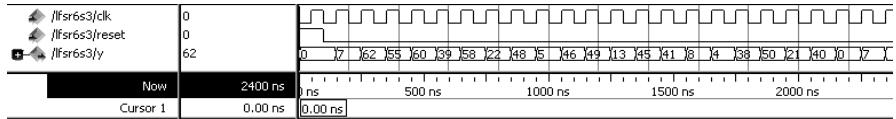


Fig. 7.20. Multistep LFSR simulation

```

      y    : OUT STD_LOGIC_VECTOR(6 DOWNTO 1);
END lfsr6s3;                                -- System output
-----
ARCHITECTURE fpga OF lfsr6s3 IS
  SIGNAL ff : STD_LOGIC_VECTOR(6 DOWNTO 1);

BEGIN

  PROCESS(clk, reset)                      -- Implement three step
  BEGIN                                     -- length-6 LFSR with xnor
    IF reset = '1' THEN                     -- Asynchronous clear
      ff <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
      ff(6) <= ff(3);
      ff(5) <= ff(2);
      ff(4) <= ff(1);
      ff(3) <= ff(5) XNOR ff(6);
      ff(2) <= ff(4) XNOR ff(5);
      ff(1) <= ff(3) XNOR ff(4);
    END IF;
  END PROCESS ;
  y <= ff;      -- Connect to I/O cell
END fpga;

```

Figure 7.20 shows a simulation of the three-step LFSR design. Comparing the simulation of this LFSR in Fig. 7.20 with the simulation of the single-step LFSR in Fig. 7.19, it can be concluded that now every third sequence value occurs. The cycle length is reduced from $2^6 - 1$ to $(2^6 - 1)/3 = 21$. The design uses 6 LEs, no embedded multiplier, and has a registered performance of $F_{max}=931.97$ MHz using the TimeQuest slow 85C model.

7.2

To implement such a multistep LFSR, we want to select the primitive polynomial that results in the lowest circuit effort, which can be computed by counting the nonzero entries in the $\mathbf{A}^k \bmod 2$ matrix, and/or the maximum fan-in for the register, which corresponds to the number of ones in each row. For a few shifts, the fan-in for the circuit from Fig. 7.18a may be advantageous. It can also be observed⁴ that if the feedback signals are close in the \mathbf{A} matrix, some entries in the \mathbf{A}^k matrix may become zero, due to

⁴ It is obviously not applicable to select the LFSR with the smallest implementation effort, because there are $\phi(2^l - 1)/l$ primitive polynomials, where $\phi(x)$ is

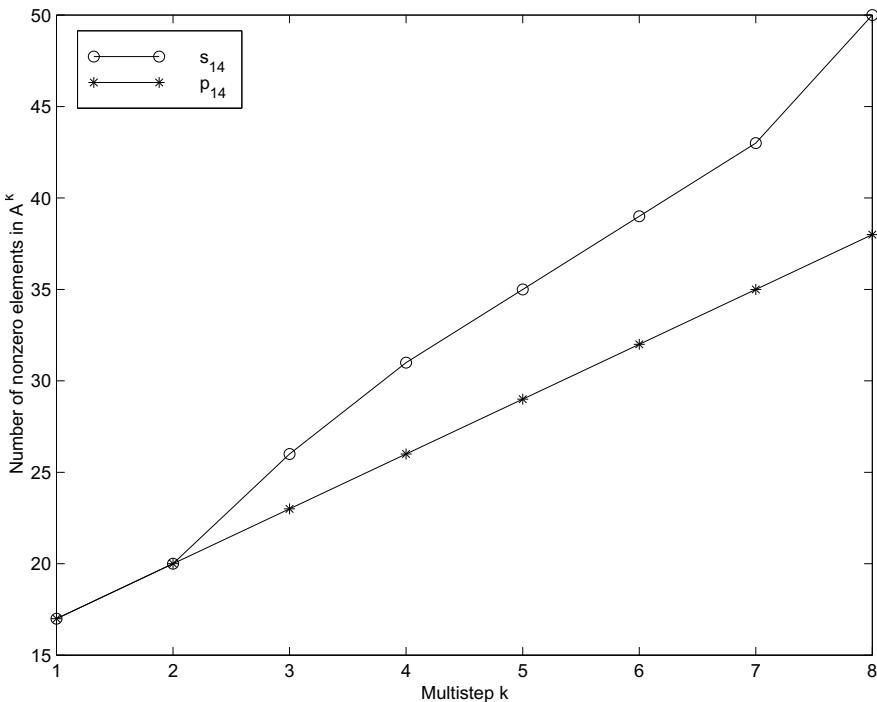


Fig. 7.21. Number of ones in A_{14}^k

the modulo 2 operations. As mentioned earlier, the two and four-tap LFSR data in Table 7.8 were therefore computed to yield the maximum sum of all taps. For the same sum, the primitive polynomial that has the larger value for the smallest tap was selected, e.g., (11, 12) is better than (10, 13). This was chosen because tap l is mandatory for the maximum-length LFSR, and the other values should be close to this tap.

If, for instance, Stanke's $s_{14,a}(x) = x^{14} + x^{12} + x^{11} + x + 1$ primitive polynomial is used, this will result in 58 entries for an $n = 8$ multistep LFSR, while if the LFSR from Table 7.8, $p_{14} = x^{14} + x^5 + x^3 + x^1 + 1$ (i.e., taps 14,13,11,9) is used, the $\mathbf{A}^8 \bmod 2$ matrix has only 35 entries (Exercise 7.6, p. 531). Fig. 7.21 shows the total number of ones for the LFSR for the two different implementations from Fig. 7.18, while Fig. 7.22 shows the maximum fan-in (i.e., the maximum needed input bit width for a LC) for this LFSR. It can be concluded from the two figures that a careful choice of the polynomial and LFSR structure can provide substantial savings. For the multistep LFSR synthesis, it can be seen from Fig. 7.22 that

the Euler function that computes the number of coprimes to x . For instance, a 16-bit register has $\phi(2^{16} - 1)/16 = 2048$ different primitive polynomials [237]!

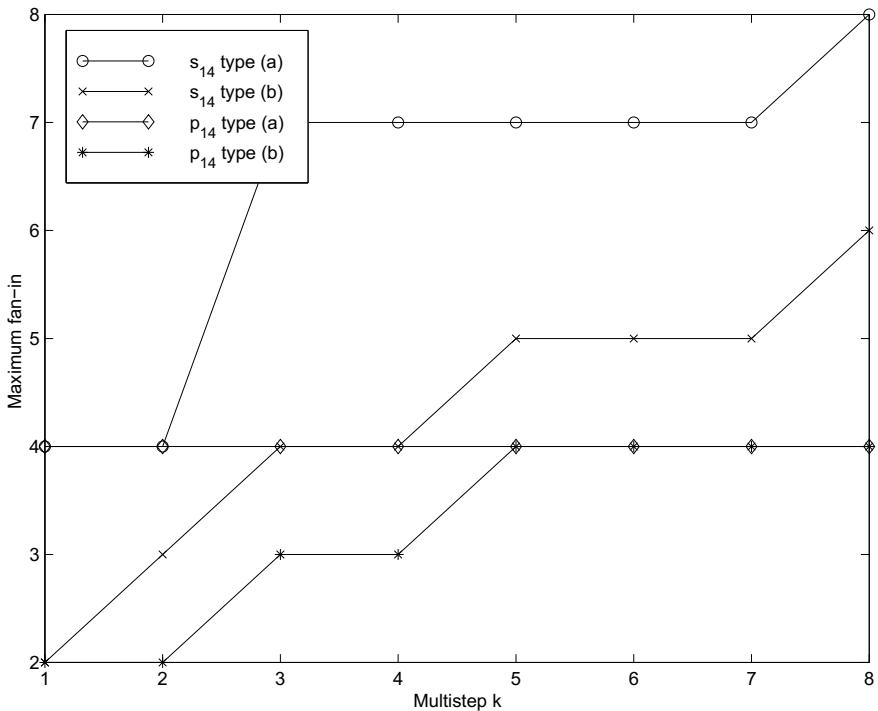


Fig. 7.22. Maximum fan-in for multistep length-14 LFSR

the LFSR of Fig. 7.18b has fewer fan-ins (i.e., smaller LC input bit width), but for longer multistep k , the effort seems similar for the primitive polynomials from Table 7.8.

Combining LFSR

An additional gain in performance in cryptographic security can be achieved if several LFSR registers are combined into one key generator. Several linear and nonlinear combinations exist [224], [223, pp. 150–173]. Meaningful for implementation effort and security are nonlinear combinations with thresholds. For a combination of three different LFSRs with length L_1 , L_2 , and L_3 the *linear complexity*, which is the equivalent length of *one* LFSR (which may be synthesized with the Berlekamp–Massey algorithm, for instance, [223, pp. 141–9]), provides

$$L_{\text{ges}} = L_1 \times L_2 + L_2 \times L_3 + L_1 \times L_3. \quad (7.23)$$

Figure 7.23 shows a realization for such a scheme.

Since the key in the selected paging format has 50 bits, a total length of $2 \times 50 = 100$ registers was chosen, and the three feedback polynomials are:

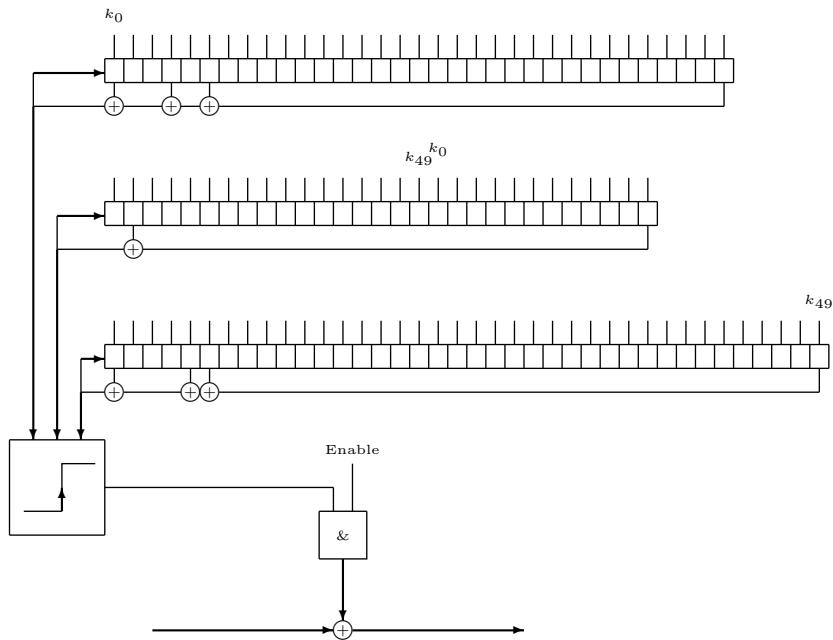


Fig. 7.23. Realization of the data-stream cipher with 3 LFSR

$$p_{33}(x) = x^{33} + x^6 + x^4 + x + 1 \quad (7.24)$$

$$p_{29}(x) = x^{29} + x^2 + 1 \quad (7.25)$$

$$p_{38}(x) = x^{38} + x^6 + x^5 + x + 1. \quad (7.26)$$

All the polynomials are *primitive*, which guarantees that the length of all three shift-register sequences gives a maximum. For the linear complexity of the combination it follows that:

$$L_1 = 33; \quad L_2 = 29; \quad L_3 = 38$$

$$L_{\text{total}} = 33 \times 29 + 33 \times 38 + 29 \times 38 = 3313.$$

After each coding the key is lost, and an additional 50 registers are needed to store the key. The 50-bit key is used twice. Table 7.9 shows the hardware resources required with Xilinx FPGAs of the Spartan-6 family.

DES-based algorithm. The data encryption standard (DES), outlined in Fig. 7.24, is typically used in a block cipher. By selecting the “output feedback mode” (OFB) it is also possible to use the modified DES in a data-stream cipher (see Fig. 7.25). The other modes (ECB, CBC, or CFB) of the DES are, in general, not applicable for communication systems, due to the “avalanche

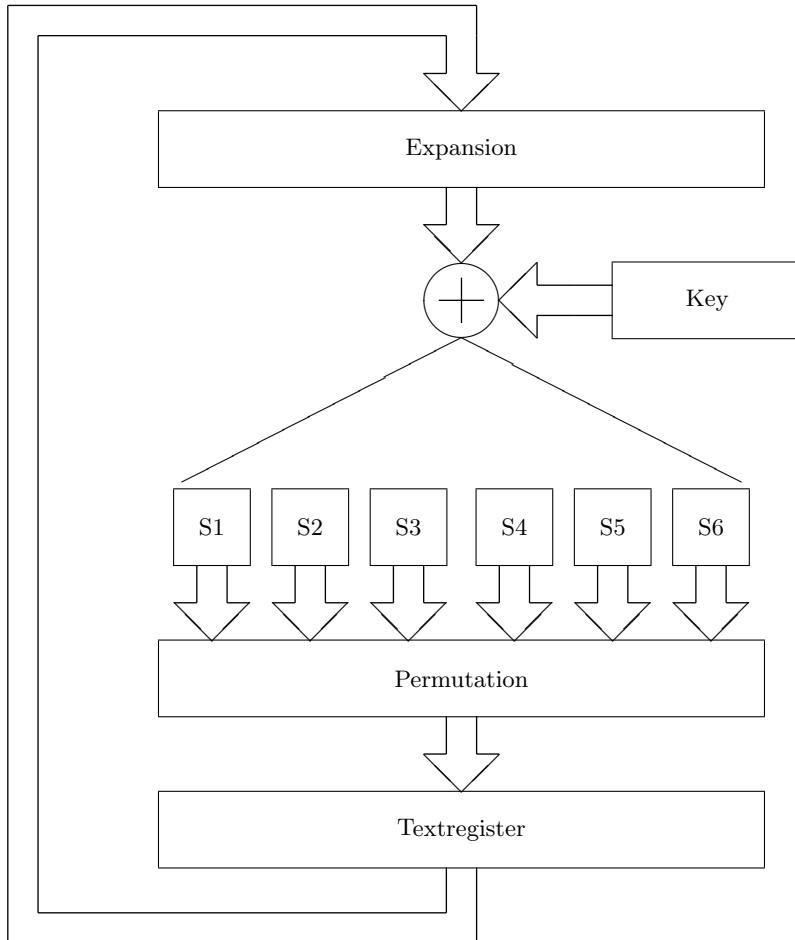


Fig. 7.24. State machine for a block encryption system (DES)

Table 7.9. Cost, measured in slices, of a Xilinx Spartan-6 FPGA

Function group	Slices
50-bit key register	6.25
100-bit shift register	12.5
Feedback	0.75
Threshold	0.25
XOR with message	0.25
Total	20

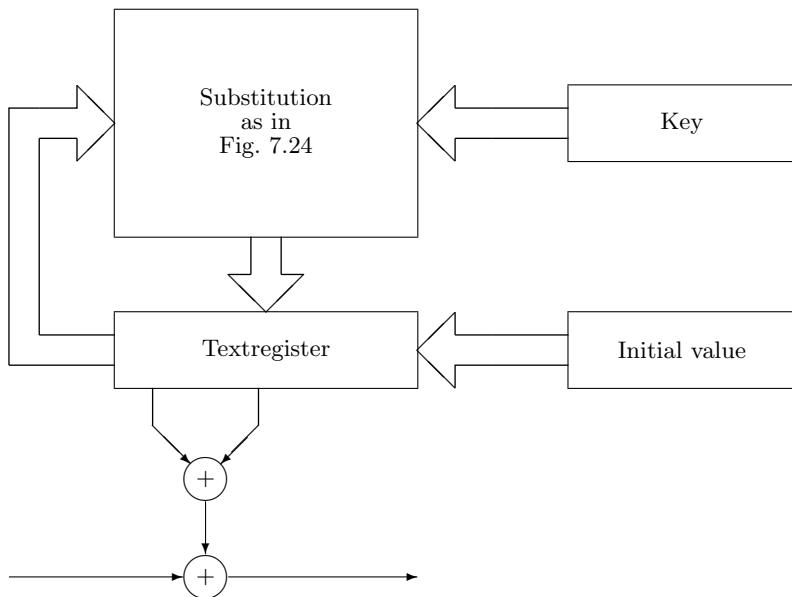


Fig. 7.25. Block cipher in the OFB-mode used as data-stream cipher

effect": A single-bit error in the transmission will alter approximately 50% of all bits in a block.

We will review the principles of the DES algorithm and then discuss suitable modifications for FPGA implementations.

The DES comprises a finite state machine translating plaintext blocks into ciphertext blocks. First the block to be substituted is loaded into the state register (32 bits). Next it is expanded (to 48 bits), combined with the key (also 48 bits) and substituted in eight $6 \rightarrow 4$ bit-width S-boxes. Finally, permutations of single bits are performed. This cycle may be (if desired, with a changing key) applied several times. In the DES, the key is usually shifted one or two bits so that after 16 rounds the key is back in the original position. Because the DES can therefore be seen as an iterative application of the Feistel cipher (shown in Fig. 7.26), the S-boxes must not be invertible. To simplify an FPGA realization some modifications are useful, such as a reduction of the length of the state register to 25 bits. No expansion is used. Use the final permutations as listed in Table 7.10.

Because most FPGAs only have four to five input look-up tables (LUTs), S-boxes with five inputs have been designed, as displayed in Table 7.11.

Although the intention was to use the OFB mode only, the S-boxes in Table 7.12 were generated in such a manner that they can be inverted. The modified DES may therefore also be used as a normal block cipher (electronic code book).

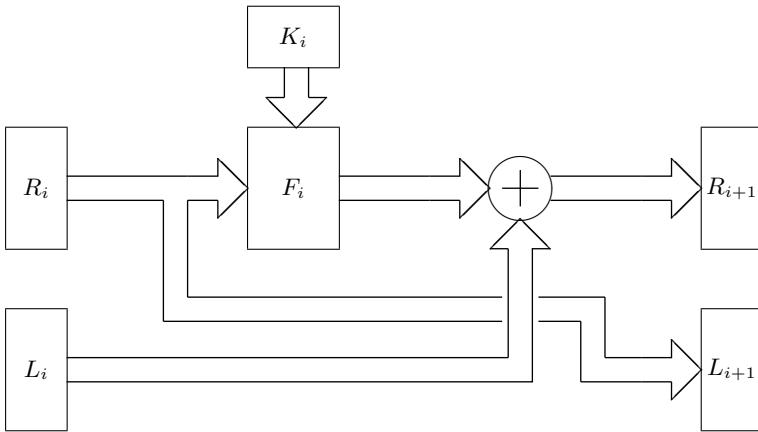


Fig. 7.26. Principle of the Feistel network

Table 7.10. Table for permutation

From bit no.	0	1	2	3	4	5	6	7	8	9	10	11	12
To bit no.	20	4	5	10	15	21	0	6	11	16	22	1	7
From bit no.	13	14	15	16	17	18	19	20	21	22	23	24	
To bit no.	12	17	23	2	8	13	18	24	3	9	14	19	

A reasonable test for S-boxes is the dependency matrix. This matrix shows, for every input/output combination, the probability that an output bit changes if an input bit is changed. With the avalanche effect the ideal probability is $1/2$. Table 7.12 shows the dependency matrix for the new five S-boxes. Instead of the probability, the table shows the absolute number of occurrences. Since there are $2^5 = 32$ possible input vectors for each S-box, the ideal value is 16. A random generator was used to generate the S-boxes. The reason that some values differ much from the ideal 16 may lie in the desired inversion.

The hardware effort of the DES-based algorithm is summarized in Table 7.13.

Cryptographic performance comparison. We will next discuss the cryptographic performance analysis of the LFSR- and DES-based algorithms. Several security tests have been defined and the following comparison shows the two most interesting (the others do not show clear differences between the two schemes). For both tests, 100 random keys were generated.

- 1) Using different keys, the generated sequences were analyzed. In each random key, one bit was changed and the number of bit changes in the

Table 7.11. The five new designed substitution boxes (S-boxes)

Input	Box 1	Box 2	Box 3	Box 4	Box 5
0	1E	F	14	19	6
1	13	1	1D	14	E
2	14	13	16	D	1A
3	1	1F	B	4	3
4	1A	19	5	1C	B
5	1B	1C	E	1A	1E
6	E	12	8	1E	0
7	B	11	F	1	2
8	D	8	4	C	1D
9	10	7	C	F	C
A	3	1B	1E	1B	18
B	0	0	13	1D	17
C	4	1A	10	5	1
D	6	C	1	15	15
E	A	1D	18	E	1B
F	17	2	17	13	9
10	19	B	1C	17	19
11	16	1E	A	9	A
12	7	18	1B	3	4
13	1C	D	3	10	14
14	1D	5	19	A	13
15	5	14	D	16	11
16	2	15	0	12	10
17	1F	9	2	1F	12
18	F	3	15	B	5
19	11	10	6	2	F
1A	C	6	7	6	8
1B	18	17	12	18	16
1C	9	4	1F	11	1C
1D	15	16	1A	8	7
1E	8	E	9	7	D
1F	12	A	11	0	1F

plaintext was recorded. On average, about 50% of the bits should be inverted (avalanche effect).

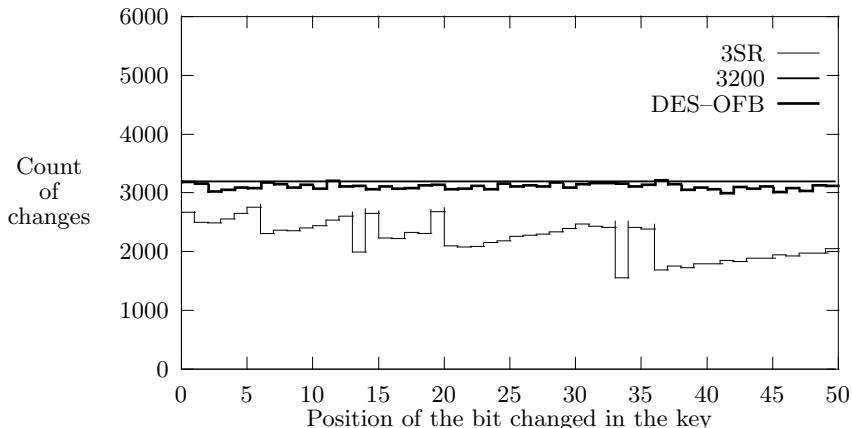
Table 7.12. Dependency matrix for the five substitution boxes (ideal value is 16)

Box 1					Box 2					Box 3				
20	12	20	20	20	20	16	20	12	20	20	12	16	16	16
12	20	12	16	16	20	20	20	16	16	16	20	16	16	16
12	16	16	12	8	12	20	20	16	8	16	16	20	12	12
16	16	20	12	16	16	24	12	16	12	16	8	12	16	20
20	16	20	12	12	16	20	16	20	20	20	12	12	20	12

Box 4					Box 5				
20	16	20	20	16	12	20	8	12	20
12	16	12	16	20	20	12	16	24	20
20	16	16	20	16	16	12	12	20	16
20	16	16	20	24	16	20	16	20	12
16	12	28	20	16	12	16	16	12	24

Table 7.13. Hardware effort of the modified DES based algorithm

Function group	Slices
25-bit key register	3.125
25-bit additions	6.25
25-bit state register	3.125
Five S-boxes 5→5	.75
Permutation	0
25-bit initialization vector	3.125
Multiplex: Initialization vector/S-box	3.125
XOR with message	0.25
Total	19.75

**Fig. 7.27.** Results of test 1

- 2) Similar to Test 1, but this time the number of changes in the output sequence were analyzed, depending on the changed position of the key. Again, 50% of the bits should change in sign.

For both tests, plaintext with 64-bit length were used (see again Fig. 7.17, p. 495). The plaintext is arbitrary. For Test 1, all variations over each individual key position were accumulated. For Test 2, all changes depending on the position in the output sequence were accumulated. This test was performed for 100 random keys. For Test 1, the ideal value is $64 \times 0.5 \times 100 = 3200$ and for Test 2 the optimal value is $50 \times 0.5 \times 100 = 2500$. Figures 7.27 and 7.28 display the results. They clearly show that the DES-OFB scheme is much more sensitive to changes in the key than is the scheme with three LFSRs. The conclusion from Test 2 is that the SR scheme needs about 32 steps until a

change in the key will affect the output sequence. For the DES-OFB scheme, only the first four samples differ considerably from the ideal value of 2500.

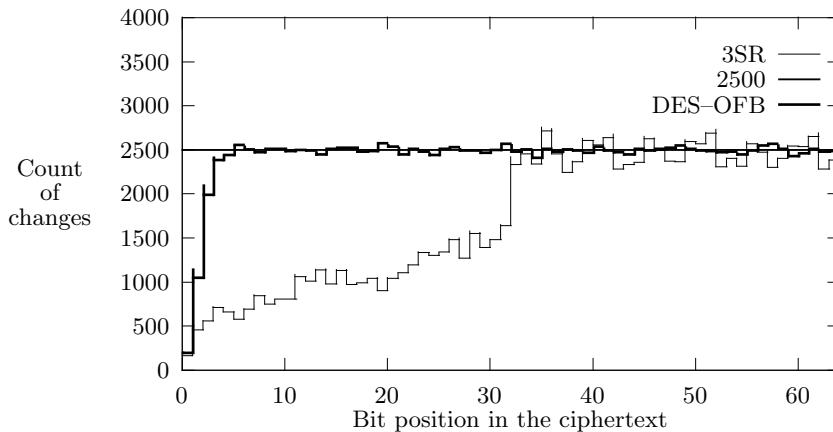


Fig. 7.28. Results of test 2

Due to the superior test results, the DES-OFB scheme may be preferred over the LFSR scheme.

A final note on encryption security. In general, it is not easy to conclude that an encryption system is secure. Besides the fact that a key may be stolen,

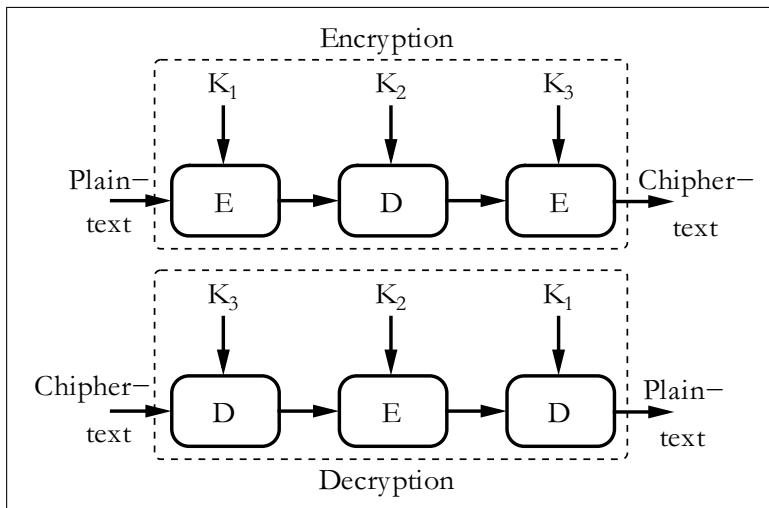


Fig. 7.29. Triple DES. (K_l = keys; E=single Encryption; D=single Decryption)

the fact that a fast crack algorithm is not *now* known does not *prove* that there are no such fast algorithms. Differential power attack algorithms, for instance, recently showed how to explore weakness in the implementation rather than a weakness in the algorithms itself [238]. There is also the problem of a “brute force attack” using more powerful computers and/or parallel attacks. A good example is the 56-bit key DES algorithm, which was the standard for many years but was finally declared insecure in 1997. The DES was first cracked by a network of volunteer computer owners on the Internet, which cracked the key in 39 days. Later, in July 1997, the Electronic Frontier Foundation (EFF) finished the design of a cracker machine. It has been documented in a book [239], including all schematics and software source code, which can be downloaded from <http://www.eff.org/>. This cracker machine performs an exhaustive key search and can crack any 56-bit key in less than five days. It was built out of custom chips, each of which has 24 cracker units. Each of the 29 boards used consists of 64 “Deep Crack” chips, i.e., a total of 1856 chips, or 44 544 units, are in use. The system cost was \$250,000. When DES was introduced in 1977 the system costs were estimated at \$20 million, which corresponds to about \$40 million today. This shows a good approximation to “Moore’s law,” which says that every 18 months the size or speed or price of microprocessors improves by a factor 2. From 1977 to 1998 the price of such a machine should drop to $40 \times 10^6 / 2^{22/1.5} \approx$ fifteen hundred dollars, i.e., it should be affordable to build a DES cracker today (as was proven by the EFF).

Therefore the 56-bit DES is no longer secure, but it is now common to use triple DES, as displayed in Fig. 7.29, or other 128-bit key systems. Table 7.14 shows that these systems seem to be secure for the next few years. The EFF cracker, for instance, today will need about 5×2^{112} days, or 7×10^{31} years, to crack the triple DES.

The first column in Table 7.14 is the commonly used abbreviations for the algorithms. The second and third columns contain the typical parameters of the algorithm. Symmetric algorithms (designated in the fourth column with an “s”) are usually based on Feistel’s algorithm, while asymmetric algorithms can be used in a public/private key system. The last column displays the name of the developer and the year the algorithm was first published.

7.2 Modulation and Demodulation

For a long time the goal of communications system design was to realize a fully digital receiver, consisting of only an antenna and a fully programmable circuit with digital filters, demodulators and/or decoders for error correction and cryptography on a single programmable chip. With today’s FPGA gate count above one million gates this has become a reality. “FPGAs will clearly be a key technology for communication systems well into the 21st century”

Table 7.14. Encryption algorithms [240]

Algorithm	Key size (bits)	Mathematical operations/ principle	Symmetry	Developed by (year)
DES	56	XOR, fixed S-boxes	s	IBM (1977)
Triple DES	122 – 168	XOR, fixed S-boxes	s	
AES	128 – 256	XOR, fixed S-boxes	s	Daemen/Rijmen (1998)
RSA	variable	Prime factors	a	Rivest/Shamir/ Adleman (1977)
IDEA	128	XOR, add., mult.	s	Massey/Lai (1991)
Blowfish	< 448	XOR, add. fixed S-boxes	s	Schneider (1993)
RC5	< 2048	XOR, add., rotation	s	Rivest (1994)
CAST-128	40 – 128	XOR, rotation, S-boxes	s	Adams/Tavares (1997)

as predicted by Carter [241]. In this section, the design and implementation of a communication system is developed in the context of FPGAs.

7.2.1 Basic Modulation Concepts

A basic communication system transmits and receives information broadcast over a carrier frequency, say f_0 . This carrier is *modulated* in amplitude, frequency or phase, proportional to the signal $x(t)$ being transmitted. Figure 7.30 shows a modulated signal for a binary transmission. For binary transmission, the modulations are called amplitude shift keying (ASK), phase shift keying (PSK), and frequency shift keying (FSK).

In general, it is more efficient to describe a (real) modulated signal with a projection of a rotating arrow on the horizontal axis, according to

$$\begin{aligned} s(t) &= \Re \left\{ A(t) e^{j(2\pi f_0 t + \Delta\phi(t) + \phi_0)} \right\} \\ &= A(t) \cos(2\pi f_0 t + \Delta\phi(t) + \phi_0), \end{aligned} \quad (7.27)$$

where ϕ_0 is a (random) phase offset, $A(t)$ describes the part of the amplitude envelope, and $\Delta\phi(t)$ describes the frequency- or phase-modulated component, as shown in Fig. 7.31. As can be seen from (7.27), AM and PM/FM can be used separately to transmit different signals.

An efficient solution (that does not require large tables) for realizing universal modulator is the CORDIC algorithm discussed in Chap. 2 (p. 131).

The CORDIC algorithm is used in the rotation mode, i.e., it is a coordinate converter from $(R, \theta) \rightarrow (X, Y)$. Figure 7.32 shows the complete modulator for AM, PM, and FM.

To implement amplitude modulation, the signal $A(t)$ is directly connected with the radius R input of the CORDIC. In general, the CORDIC algorithm in rotation mode has an attendant linear increase in the radius. This corresponds to a change in the gain of an amplifier and need not be taken into consideration for the AM scheme. When the linear increased radius (factor 1.6468, see Table 2.1, p. 59), is not desired, it is possible either to scale the input or the output by $1/1.6468$ with a constant coefficient multiplier.

The phase of the transmitted signal $\theta = 2\pi f_0 t + \Delta\phi(t)$ must also be computed. To generate the constant carrier frequency, a linearly increasing phase signal according to $2\pi f_0 t$ must be generated, which can be done with an accumulator. If FM should be generated, it is possible to modify f_0 by Δf , or to use a second accumulator to compute $2\pi \Delta f t$, and to add the results of the two accumulators. For the PM signal, a constant offset (not increasing in time) is added to the phase of the signal. These phase signals are added and applied to the angle input z or θ of the CORDIC processor. The Y register is set to zero at the beginning of the iterations.

The following example demonstrates a fully pipelined version of the CORDIC modulator.

Example 7.3: Universal Modulator using CORDIC

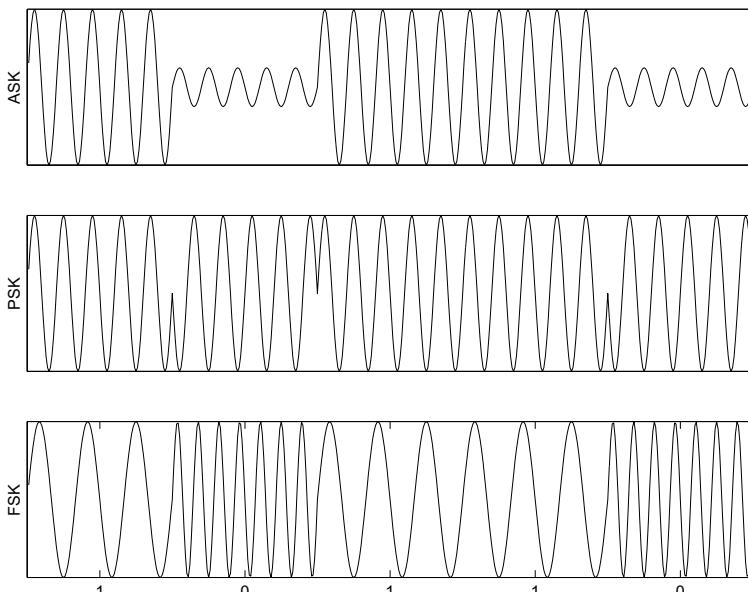
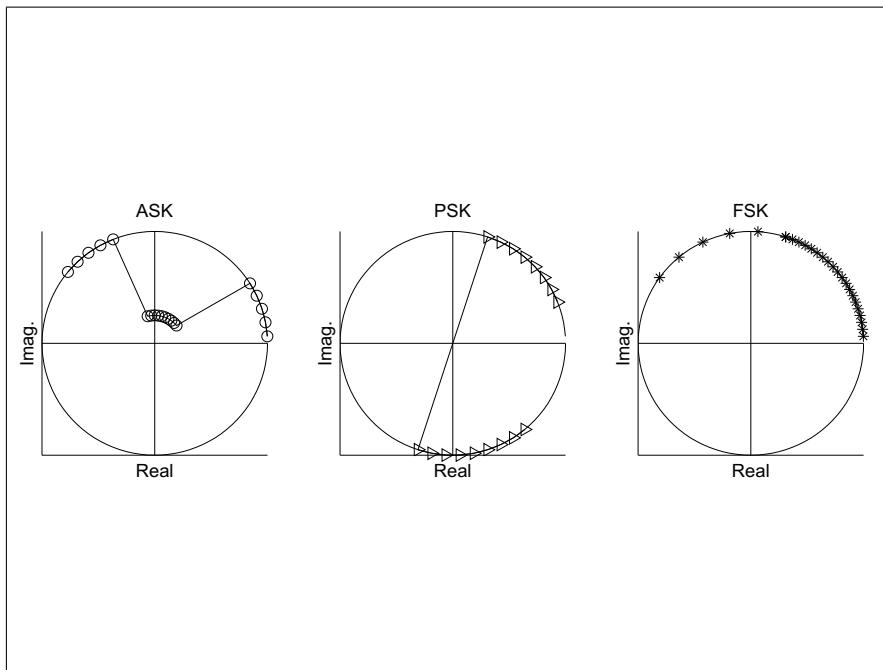
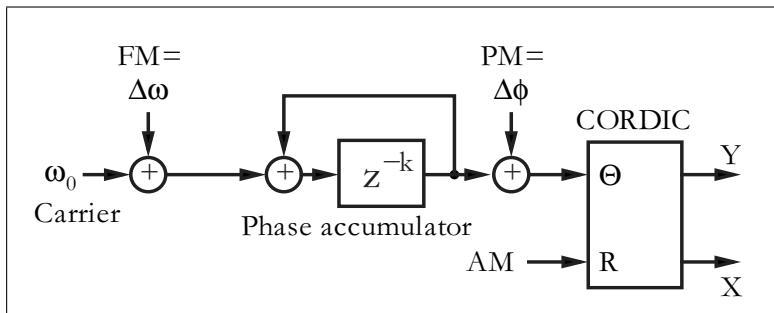


Fig. 7.30. ASK, PSK, and FSK modulation

**Fig. 7.31.** Modulation in the complex plan**Fig. 7.32.** Universal modulator using CORDIC

A universal modulator for AM, PM, and FM according to Fig. 7.32, can be designed with the following VHDL code⁵ of the CORDIC part:

```

PACKAGE n_bit_int IS      -- User defined types
  SUBTYPE S9 IS INTEGER RANGE -256 TO 255;
  TYPE A0_3S9 IS ARRAY (0 TO 3) OF S9;
```

⁵ The equivalent Verilog code `ammod.v` for this example can be found in Appendix A on page 859. Synthesis results are shown in Appendix B on page 881.

```

END n_bit_int;

LIBRARY work; USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL; USE ieee.std_logic_arith.ALL;
-----
ENTITY ammod IS          -----> Interface
    PORT (clk      : IN STD_LOGIC;  -- System clock
          reset    : IN STD_LOGIC;  -- Asynchronous reset
          r_in     : IN S9;        -- Radius input
          phi_in   : IN S9;        -- Phase input
          x_out    : OUT S9;       -- x or real part output
          y_out    : OUT S9;       -- y or imaginary part
          eps      : OUT S9);    -- Error of results
END ammod;
-----
ARCHITECTURE fpga OF ammod IS

BEGIN

PROCESS(clk, reset, r_in, phi_in) --> Behavioral style
    VARIABLE x, y, z : A0_3S9; -- Register arrays
BEGIN
    IF reset = '1' THEN -- Asynchronous clear
        FOR k IN 0 TO 3 LOOP
            x(k) := 0; y(k) := 0; z(k) := 0;
        END LOOP;
        x_out <= 0; eps <= 0; y_out <= 0;
    ELSIF rising_edge(clk) THEN
        -- Compute last value first
        x_out <= x(3);           -- in sequential statements !!
        eps    <= z(3);
        y_out <= y(3);

        IF z(2) >= 0 THEN           -- Rotate 14 degrees
            x(3) := x(2) - y(2) /4;
            y(3) := y(2) + x(2) /4;
            z(3) := z(2) - 14;
        ELSE
            x(3) := x(2) + y(2) /4;
            y(3) := y(2) - x(2) /4;
            z(3) := z(2) + 14;
        END IF;

        IF z(1) >= 0 THEN           -- Rotate 26 degrees
            x(2) := x(1) - y(1) /2;
            y(2) := y(1) + x(1) /2;
            z(2) := z(1) - 26;
        ELSE
            x(2) := x(1) + y(1) /2;
            y(2) := y(1) - x(1) /2;
            z(2) := z(1) + 26;
        END IF;
    END IF;
END;

```

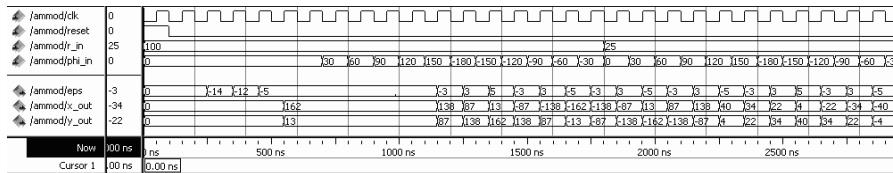


Fig. 7.33. Simulation of an AM modulator using the CORDIC algorithm

```

END IF;

IF z(0) >= 0 THEN
    x(1) := x(0) - y(0);
    y(1) := y(0) + x(0);
    z(1) := z(0) - 45;
-- Rotate 45 degrees
ELSE
    x(1) := x(0) + y(0);
    y(1) := y(0) - x(0);
    z(1) := z(0) + 45;
END IF;

IF phi_in > 90      THEN
    x(0) := 0;
    y(0) := r_in;
    z(0) := phi_in - 90;
-- Test for |phi_in| > 90
-- Rotate 90 degrees
-- Input in register 0
ELSIF phi_in < -90 THEN
    x(0) := 0;
    y(0) := - r_in;
    z(0) := phi_in + 90;
ELSIF
    x(0) := r_in;
    y(0) := 0;
    z(0) := phi_in;
END IF;
END IF;
END PROCESS;

END fpga;

```

Figure 7.33 reports the simulation of an AM signal. Note that the Altera simulation allows you to use signed data, rather than unsigned binary data (where negative values have a 512 offset). A pipeline delay of four steps is seen and the value 100 is enlarged by a factor of 1.6. A switch in radius r_{in} from 100 to 25 results in the maximum value x_{out} dropping from 163 to 42. The CORDIC modulator runs at a registered performance of $F_{max}=197.39$ MHz using the TimeQuest slow 85C model and uses 264 LEs and no embedded multiplier.

Demodulation may be *coherent* or *incoherent*. A coherent receiver must recover the unknown carrier phase ϕ_0 , while an incoherent one does not need to do so. If the receiver uses an intermediate frequency (IF) band, this type

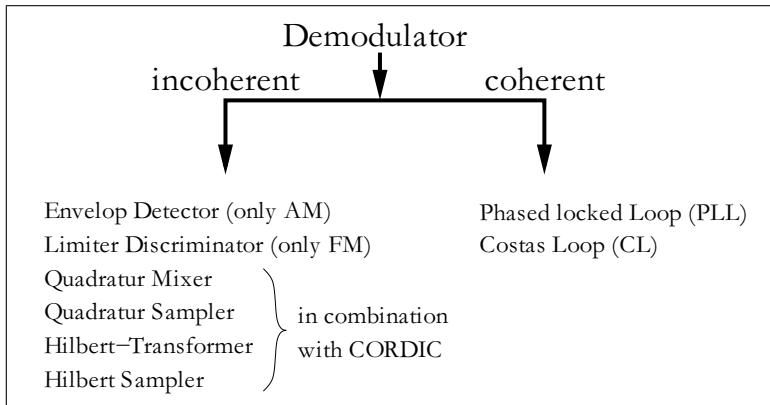


Fig. 7.34. Coherent and incoherent demodulation schemes

of receiver is called a superhet or double superhet (two IF bands) receiver. IF receivers are also sometimes called *heterodyne* receivers. If no IF stages are employed, a *zero IF* or *homodyne* receiver results. Figure 7.34 presents a systematic overview of the different types of receivers. Some of the receivers can only be used for one modulation scheme, while others can be used for multiple modes (e.g., AM, PM, and FM). The latter is called a universal receiver. We will first discuss the incoherent receiver, and then the coherent receiver.

All receivers use intensive filters (as discussed in Chaps. 3 and 4), in order to select only the signal components of interest. In addition, for heterodyne receivers, filters are needed to suppress the mirror frequencies, which arise from the frequency shift

$$s(t) \times \cos(2\pi f_m t) \longleftrightarrow S(f + f_m) + S(f - f_m). \quad (7.28)$$

7.2.2 Incoherent Demodulation

In an incoherent demodulation scheme, it is assumed that the exact carrier frequency is known to the receiver, but the initial phase ϕ_0 is not.

If the signal component is successfully selected with digital or analog filtering, the question arises whether only one demodulation mode (e.g., AM or FM) or universal demodulator is needed. An incoherent AM demodulator can be as simple as a full or half-wave rectifier and an additional lowpass filter. For FM or PM demodulation, only the *limiter/discriminator* type of demodulator is an efficient implementation. This demodulator builds a threshold of the input signal to limit the values to ± 1 , and then basically “measures” the distance between the zero crossings. These receivers are easily implemented with FPGAs but sometimes produce 2π jumps in the phase signal (called “clicks” [242, 243]). There are other demodulators with better performance.

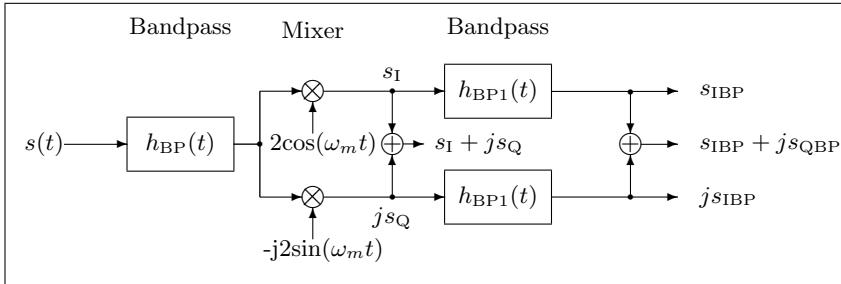


Fig. 7.35. Generation of I- and Q-phase using quadrature scheme

We will focus on universal receivers using in-phase and quadrature components. This type of receiver basically inverts the modulation scheme relative to (7.27) from p. 510. In a first step we have to compute, from the received cosines, components that are “in-phase” with the sender’s sine components (which are in quadrature to the carrier, hence the name *Q* phase). These I and Q phases are used to reconstruct the arrow (rotating with the carrier frequency) in the complex plane. Now, the demodulation is just the inversion of the circuit from Fig. 7.32. It is possible to use the CORDIC algorithm in the vectoring mode, i.e., a coordinate conversion $X, Y \rightarrow R, \theta$ with $I = X$ and $Q = Y$ is used. Then the output R is directly proportional to the AM portion, and the PM/FM part can be reconstructed from the θ signal, i.e., the Z register.

A difficult part of demodulation is I/Q generation, and typically two methods are used: a quadrature scheme and a Hilbert transform.

In the quadrature scheme the input signal is multiplied by the two mixer signals, $2\cos(2\pi f_m t)$ and $-j2\sin(2\pi f_m t)$. If the signals in the IF band $f_{IF} = f_0 - f_m$ are now selected with a filter, the complex sum of these signals is then a reconstruction of the complex rotating arrow. Figure 7.35 shows this scheme while Fig. 7.36 displays an example of the I/Q generation. From the spectra shown in Fig. 7.36 it can be seen that the final signal has no negative spectral components. This is typical for this type of incoherent receiver and these signals are called *analytic*.

To decrease the effort for the filters, it is desirable to have an IF frequency close to zero. In an analog scheme (especially for AM) this often introduces a new problem, that the amplifier drifts into saturation. But for a fully digital receiver, such a homodyne or zero IF receiver can be built. The bandpass filters then reduce to lowpass filters. Hogenauer’s CIC filters (see Chap. 5, p. 318) are efficient realizations of these high decimation filters. Figure 7.37 shows the corresponding spectra. The real input signal is sampled at 2π . Then the signal is multiplied with a cosine signal $S_{2\cos}(e^{j\omega})$ and a sine signal $S_{-j2\sin}(e^{j\omega})$. This produces the in-phase component $S_I(e^{j\omega})$ and the quadrature component $jS_Q(e^{j\omega})$. These two signals are now combined

into a complex analytic signal $S_I + jS_Q$. After the final lowpass filtering, a decimation in sampling rate can be applied.

Such a fully digital zero IF for LF has been built using FPGA technology [244].

Example 7.4: Zero IF Receiver

This receiver has an antenna, a programmable gain adjust (AGC), and a Cauer lowpass seventh order followed by an 8-bit video A/D converter. The receiver uses eight times oversampling (0.4–1.2 MHz) for the input range from 50 to 150 kHz. The quadrature multipliers are 8×8 -bit array multipliers. Two-stage CIC filters were designed with 24- and 19-bit integrator precision, and 17- and 16-bits precision for the comb sections. The final sampling rate reduction was 64. The full design could fit on a single Spartan-6 Xilinx FPGA of the Atlys board. The following table shows the effort for the single units:

Design part	Slices
Mixer with sin/cos tables	18.5
Two CIC filters	42
State machine and PDSP interface	4.5
Frequency synthesizer	8
Total	73

For the tunable frequency synthesizer an accumulator as reference for an analog phase-locked loop (PLL) was used [4]. Figure 7.38 shows this type

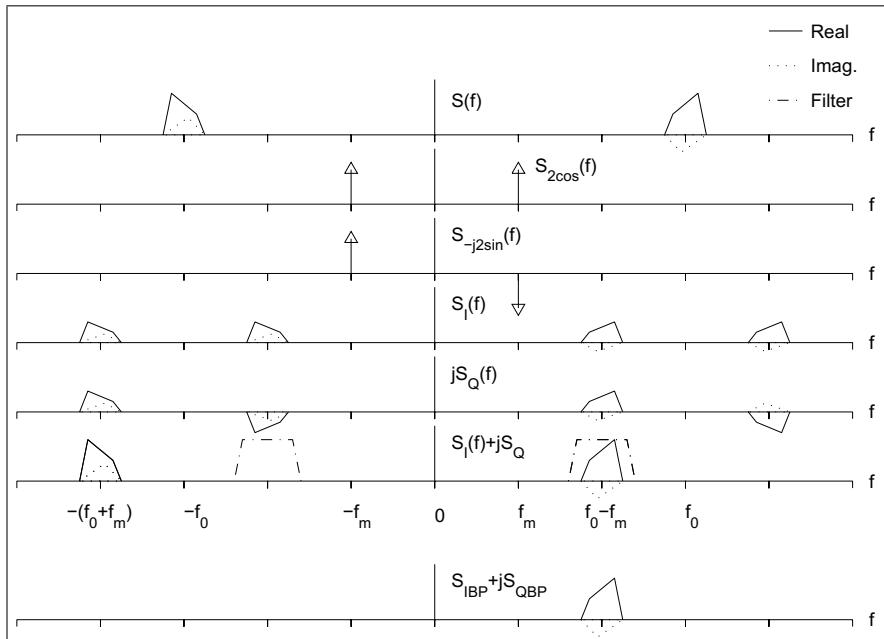


Fig. 7.36. Spectral example of the I/Q generation

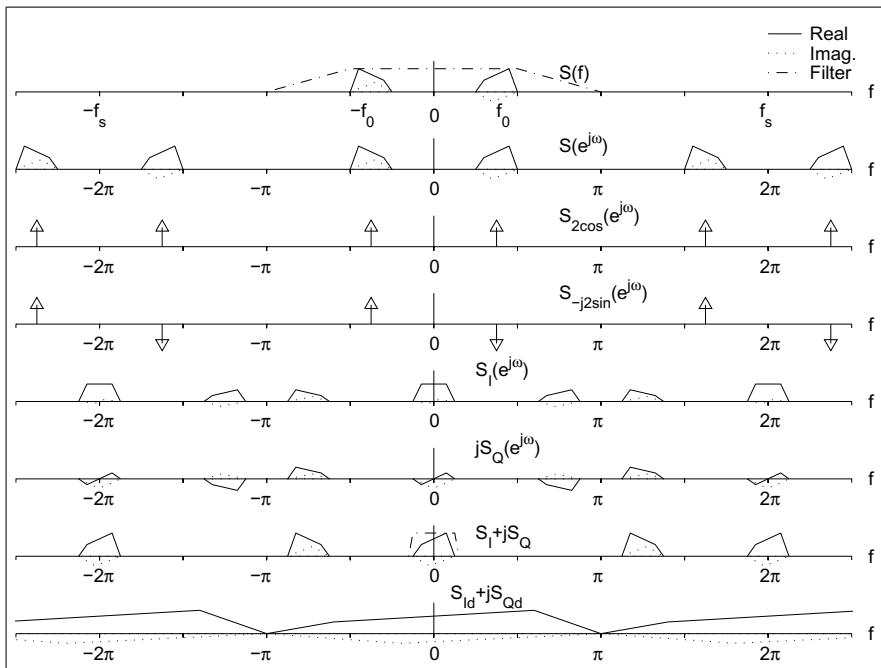


Fig. 7.37. Spectra for the zero IF receiver. Sampling frequency was 2π

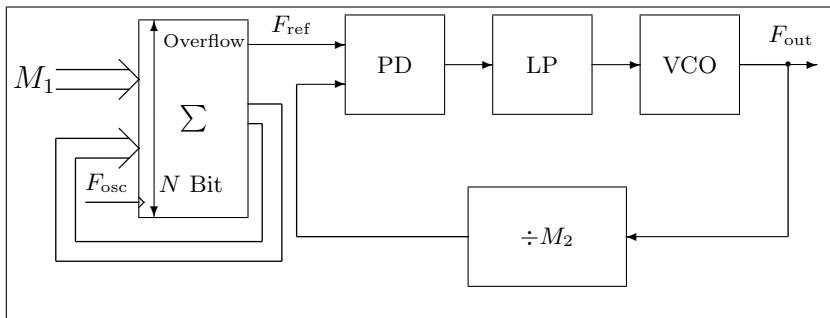


Fig. 7.38. PLL with accumulator as reference

of frequency synthesizer and Fig. 7.39 displays the measured performance of the synthesizer. The accumulator synthesizer could be clocked very high due to the fact that only the overflow is needed. A bitwise carry save adder was therefore used. The accumulator was used as a reference for the PLL that produces $F_{\text{out}} = M_2 F'_{\text{in}} = M_1 M_2 F_{\text{in}} / 2^N$.

7.4

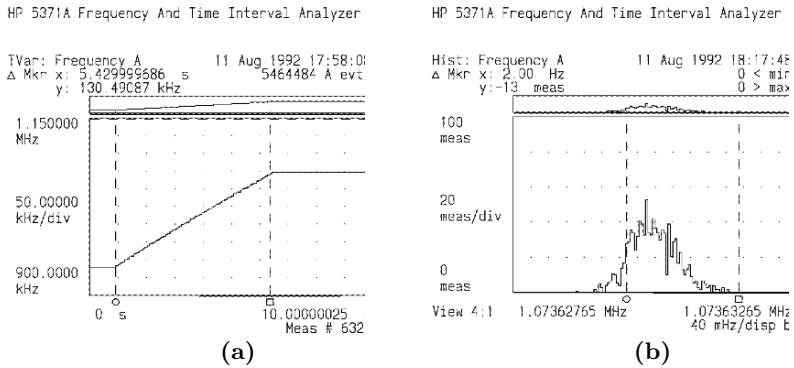


Fig. 7.39. PLL synthesizers with accumulator reference. **(a)** Behavior of the synthesizer for switching F_{out} from 900 kHz to 1.2 MHz. **(b)** Histogram of the frequency error, which is less than 2 Hz

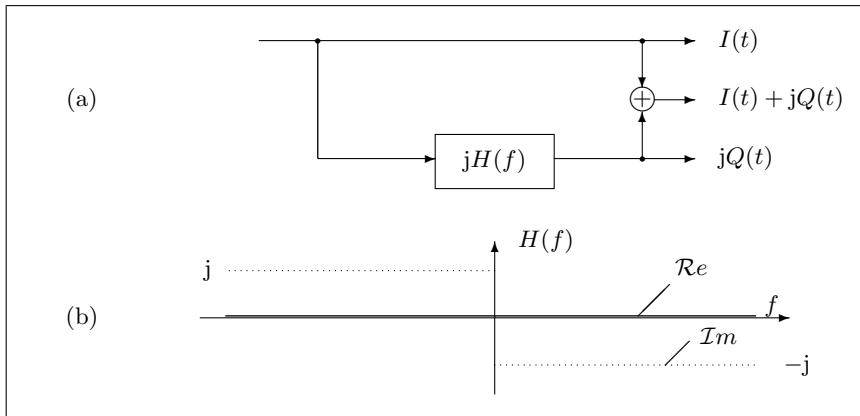


Fig. 7.40. Hilbert transformer. **(a)** Filter. **(b)** Spectrum of $H(f)$

The *Hilbert transformer* scheme relies on the fact that a sine signal can be computed from the cosine signal by a phase delay of 90° . If a filter is used to produce this Hilbert transformer, the amplitude of the filter must be one and the phase must be 90° for all frequencies. Impulse response and transfer function can be found using the definition of the Fourier transform, i.e.,

$$h(t) = \frac{1}{\pi t} \longleftrightarrow H(j\omega) = -j\gamma(\omega) = \begin{cases} j & -\infty < \omega < 0 \\ -j & 0 < \omega < \infty, \end{cases} \quad (7.29)$$

with $\gamma(\omega) = -1 \forall \omega < 0$ and $\gamma(\omega) = 1 \forall \omega \geq 0$ as the sign function. A Hilbert filter can only be approximated by an FIR filter and resulting coefficients have been reported (see, for instance, [245, 246], [186, pp. 168–174], or [88, p. 681]).

Simplification for narrowband receivers. If the input signals are narrowband signals, i.e., the transmitted bit rate is much smaller than the carrier frequency, some simplifications in the demodulation scheme are possible. In the *input sampling scheme* it is then possible to sample at the carrier rate, or at a multiple of the period $T_0 = 1/f_0$ of the carrier, in order to ensure that the sampled signals are already free of the carrier component.

The *quadrature scheme* becomes trivial if the zero IF receiver samples at $4f_0$. In this case, the sine and cosine components are elements of 0, 1 or -1, and the carrier phase is 0, 90°, 180°.... This is sometimes referred to as “complex sampling” in the literature [247, 248]. It is possible to use *undersampling*, i.e., only every second or third carrier period is evaluated by the sampler. Then the sampled signal will still be free from the carrier frequency.

The *Hilbert transformer* can also be simplified if the signal is sampled at $T_0/4$. A Hilbert sampler of first order with $Q_{-1} = 1$ or a second order type using the symmetric coefficients $Q_1 = -0.5; Q_{-1} = 0.5$ or asymmetric $Q_{-1} = 1.5; Q_{-3} = 0.5$ coefficients can be used [249].

Table 7.15. Coefficients of the Hilbert sampler

Type	Coefficients	Bit	$\Delta f/f_0$
Zero order	$Q_{-1} = 1, 0$	8	0.005069
	$Q_{-1} = 1, 0$	12	0.000320
	$Q_{-1} = 1, 0$	16	0.000020
First- order asymmetric	$Q_{-1} = 1, 5; Q_{-3} = 0.5$	8	0.032805
	$Q_{-1} = 1, 5; Q_{-3} = 0.5$	12	0.008238
	$Q_{-1} = 1, 5; Q_{-3} = 0.5$	16	0.002069
First- order symmetric	$Q_1 = -0.5; Q_{-1} = 0.5$	8	0.056825
	$Q_1 = -0.5; Q_{-1} = 0.5$	12	0.014269
	$Q_1 = -0.5; Q_{-1} = 0.5$	16	0.003584

Table 7.15 reports the three short-term Hilbert transformer coefficients and the maximum allowed frequency offset Δf of the modulation, for the Hilbert filter providing a specified accuracy.

Figure 7.41 shows two possible realizations for the Hilbert transformer that have been used to demodulate radio control watch signals [250, 251]. The first method uses three Sample & Hold circuits and the second method uses three A/D converters to build a symmetric Hilbert sampler of first order.

Figure 7.42 shows the spectral behavior of the Hilbert sampler with a direct undersampling by two.

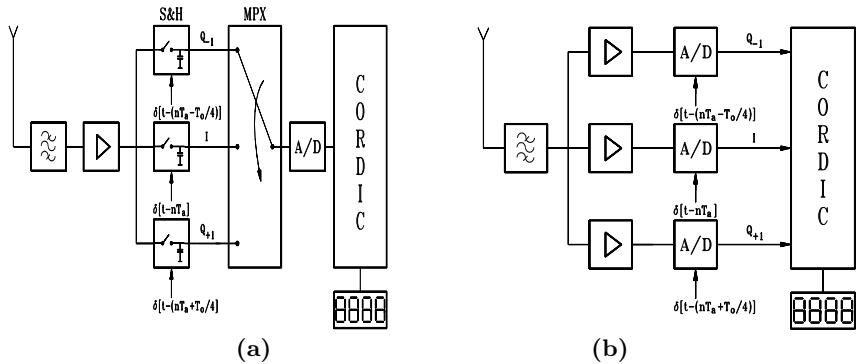


Fig. 7.41. (a) Two versions of the Hilbert sampler of first order

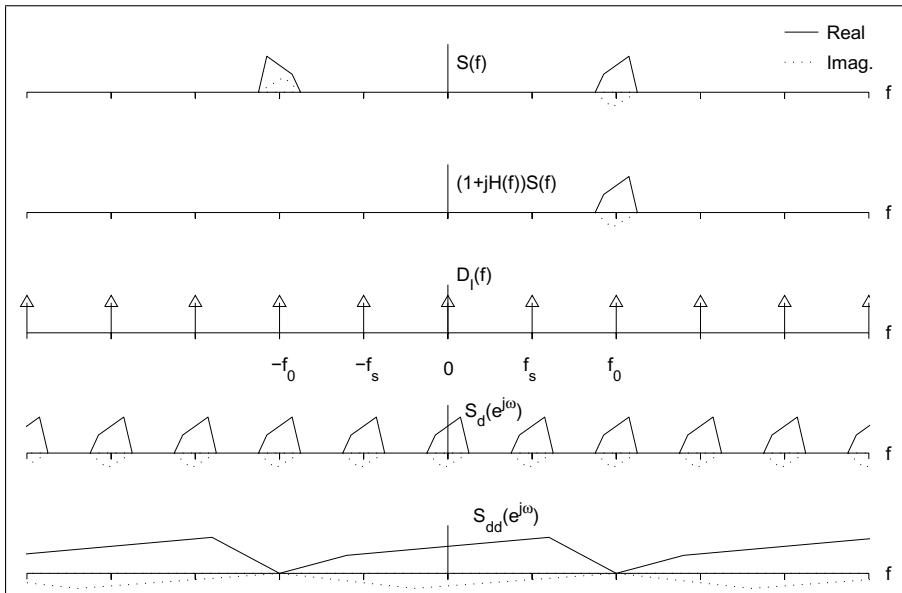


Fig. 7.42. Spectra for the Hilbert sampler with undersampling

7.2.3 Coherent Demodulation

If the phase ϕ_0 of the receiver is known, then demodulation can be accomplished by multiplication and lowpass filtering. For AM, the received signal $s(t)$ is multiplied by $2 \cos(\omega_0 t + \phi_0)$ and for PM or FM by $-2 \sin(\omega_0 t + \phi_0)$. It follows that:

AM:

$$A(t) \cos(2\pi f_0 t + \phi_0) \times 2 \cos(2\pi f_0 t + \phi_0)$$

$$= \underbrace{A(t)}_{\text{Lowpass component}} + A(t) \cos(4\pi f_0 t + 2\phi_0) \quad (7.30)$$

$$s_{\text{AM}}(t) = A(t) - A_0. \quad (7.31)$$

PM:

$$\begin{aligned} & -2 \sin(2\pi f_0 t + \phi_0) \times \cos(2\pi f_0 t + \phi_0 + \Delta\phi(t)) \\ &= \underbrace{\sin(\Delta\phi(t))}_{\text{Lowpass component}} + \cos(4\pi f_0 t + 2\phi_0 + \Delta\phi(t)) \end{aligned} \quad (7.32)$$

$$\sin(\Delta\phi(t)) \approx \Delta\phi(t) \quad (7.33)$$

$$s_{\text{PM}}(t) = \frac{1}{\eta} \Delta\phi(t). \quad (7.34)$$

FM:

$$s_{\text{FM}}(t) = \frac{1}{\eta} \frac{d \Delta\phi(t)}{d t}. \quad (7.35)$$

η is the so-called modulation index.

In the following we will discuss the types of coherent receivers that are suitable for an FPGA implementation. Typically, coherent receivers provide a one dB better signal-to-noise ratio than an incoherent receiver (see Fig. 7.1, p. 476). A synchronous or coherent FM receiver tracks the carrier phase of the incoming signal with a voltage-controlled oscillator (VCO) in a loop. The DC part of this voltage is directly proportional to the FM signal. PM signal demodulation requires integration of the VCO control signal, and AM demodulation requires the addition of a second mixer and a $\pi/2$ phase shifter in sequence with a lowpass filter. The risk with coherent demodulation is that for a low signal-to-noise channel, the loops may be out-of-lock, and performance will decrease tremendously.

There are two common types of coherent receiver loops: the phase-locked loop (PLL) and the Costas loop (CL). Figures 7.43 and 7.44 are block diagrams of a PLL and CL, respectively, showing the nearly doubled complexity of the CL. Each loop may be realized as an analog (linear PLL/CL) or all-digital (ADPLL, ADCL) circuit (see [252–255]). The stability analysis of these loops is beyond the scope of the book and is well covered in the literature ([256–260]). We will discuss efficient realizations of PLLs and CLs [261, 262]. The first PLL is a direct translation of an analog PLL to FPGA technology.

Linear phase-locked loop. The difference between linear and digital loops lies in the type of input signal to be processed. A linear PLL or CL uses a fast multiplier as a phase detector, providing a possibly multilevel input signal to the loop. A digital PLL or CL can process only binary input signals. (*Digital* refers to the quality of the input signal here, not to the hardware realization!)

As shown in Fig. 7.43, the linear PLL has three main blocks:

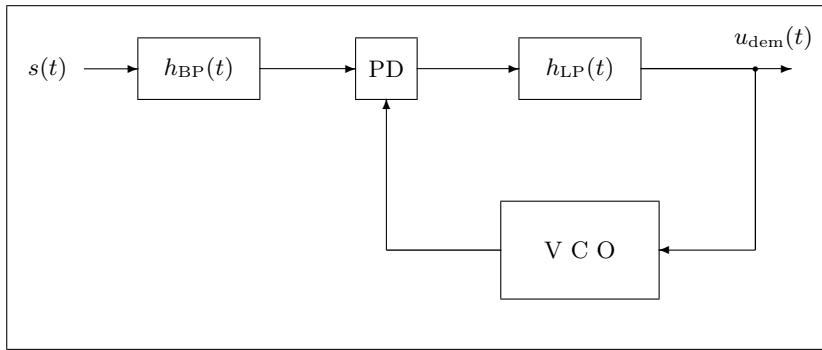


Fig. 7.43. Phase-locked loop (PLL) with (necessary) bandpass ($h_{BP}(t)$), phase detector (PD), lowpass ($h_{LP}(t)$), and voltage-controlled oscillator (VCO)

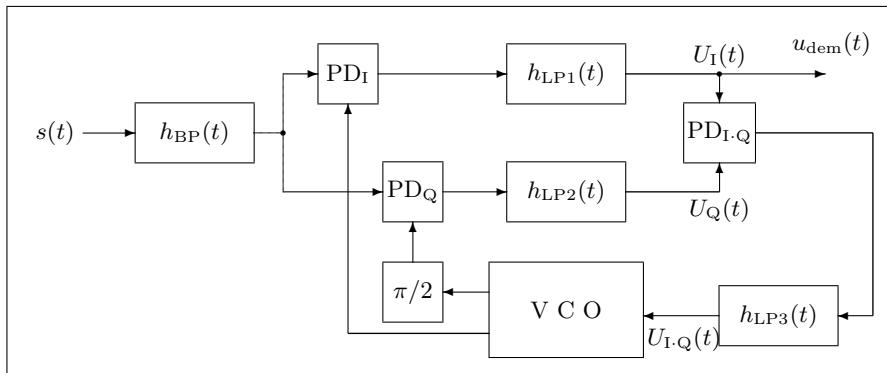


Fig. 7.44. Costas loop with (necessary) bandpass ($h_{BP}(t)$), three phase detectors (PD), three lowpass filters ($h_{LP}(t)$), and a voltage-controlled oscillator (VCO) with $\pi/2$ phase shifter

- Multiplier as phase detector
- Loop filter
- VCO

To keep the loop in-lock, a loop output signal-to-noise ratio larger than $4 = 6$ dB is required [255, p. 35]. Since the selection of typical antennas is not narrow enough to achieve this, an additional narrow bandpass filter has been added to Figs. 7.43 and 7.44 as a necessary addition to the demodulator [262]. The “cascaded bandpass comb” filter (see Table 5.4, p. 344) is an efficient example. However, the filter design is much easier if a fixed IF is used, as in the case of a superhet or double superhet receiver.

The VCO (or digitally controlled oscillator (DCO) for ADPLLs) oscillates with a frequency $\omega_2 = \omega_0 + K_0 \times U_f(t)$, where ω_0 is the resting point and K_0 the gain of the VCO/DCO. For sinusoidal input signals we have the signal

$$u_{\text{dem}}(t) = K_d \sin(\Delta\phi(t)) \quad (7.36)$$

at the output of the lowpass, where $\Delta\phi(t)$ is the phase difference between the DCO output and the bandpass-filtered input signal. For small differences, the sine can be approximated by its argument, giving $u_{\text{dem}}(t)$ proportional to $\Delta\phi(t)$ (the loop stays in-lock). If the input signal has a very sudden phase discontinuity, the loop will go out-of-lock. Figure 7.45 shows the different operation areas of the loop. The hold-in range $\omega_0 \pm \Delta\omega_H$ is the static operation limit (useful only with a frequency synthesizer). The lock-in range is the area where the PLL will lock-in within a single period of the frequency difference $\omega_1 - \omega_2$. Within the pull-in range, the loop will lock-in within the capture time T_L , which may last more than one period of $\omega_1 - \omega_2$. The pull-out range is the maximum frequency jump the loop can sustain without going out-of-lock. $\omega_0 \pm \Delta\omega_{PO}$ is the dynamic operation limit used in demodulation. There is much literature optimizing PD, loop filter and the VCO gain; see [256–260].

The major advantage of the linear loop over the digital PLL and CL is its noise-reduction capability, but the fast multiplier used for the PD in a linear PLL has a particularly high hardware cost, and this PD has an unstable rest point at $\pi/2$ phase shift ($-\pi/2$ is a stable point), which impedes lock-in. Table 7.16 estimates the hardware cost in Slices of a linear PLL, using the same functional blocks as the 8-bit incoherent receiver (see Example 7.4, p. 517). Using a hardware multiplier in multiplex for AM and PM demodulation, the right-hand column reduces the circuit's cost by 58 CLBs (≈ 14.5 Spartan-6 slices), allowing it to fit into the smallest Spartan-6 Xilinx device.

A comparison of these costs to those of an incoherent receiver, consuming 292 CLBs (≈ 73 Spartan-6 slices) without CORDIC demodulation and 367.5 CLBs (≈ 91.875 Spartan-6 slices) with an additional CORDIC processor [77], shows a slight improvement in the linear PLL realization. If only FM and PM demodulation are required, a digital PLL or CL, described in the next two sections, can reduce complexity dramatically.

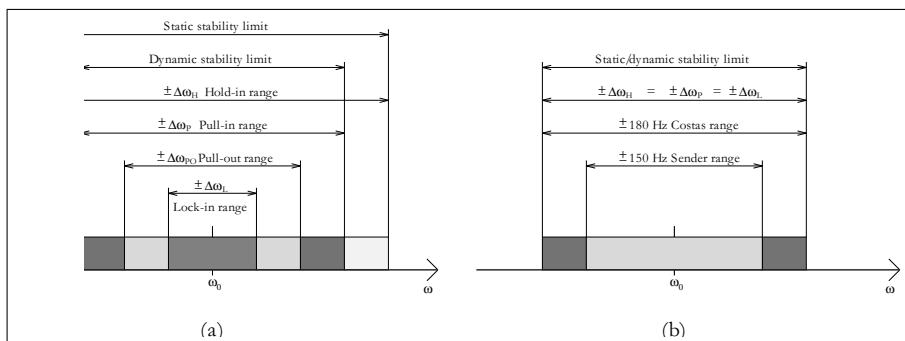


Fig. 7.45. (a) Operation area PLL/CL. (b) Operation area of the CL of Fig. 7.48

Table 7.16. Cost in Slices of a linear PLL universal demodulator in a Xilinx Spartan-6 FPGA

Function group	FM only	FM, AM, and PM
Phase detector (8×8 -bit multiplier)	16.25	18
Loop filter (two-stage CIC)	21	42
Frequency synthesizer	8.5	8.5
DCO ($N/N + K$ divider, sin/cos table)	4	4.5
PDSP Interface	3.75	3.75
Total	53.5	76.75

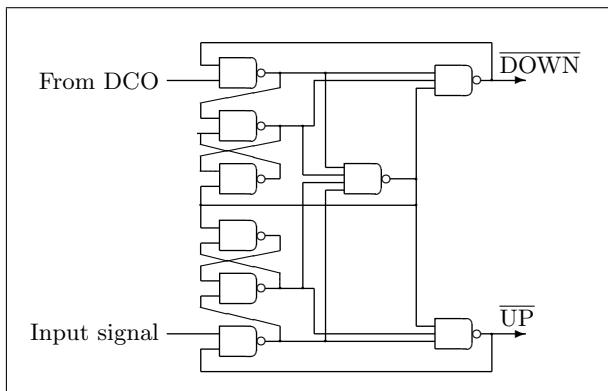
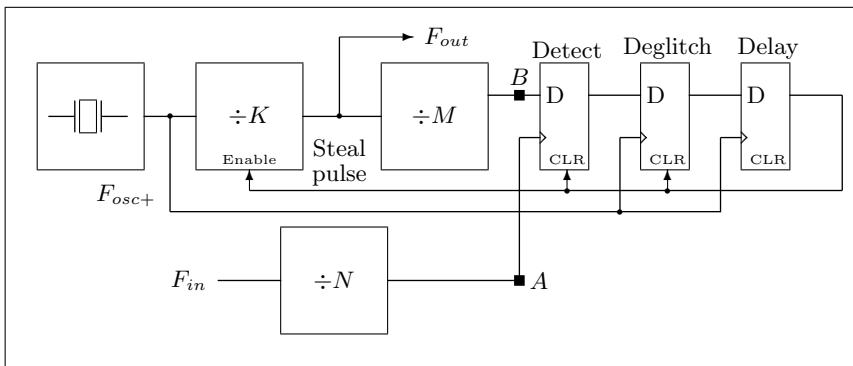


Fig. 7.46. Phase detector [22, Chap. 8 p. 127]

These designs were developed to demodulate WeatherFAX pictures, which were transmitted in central Europe by the low-frequency radio stations DCF37 and DCF54 (Carrier 117.4 kHz and 134.2 kHz; frequency modulation F1C: ± 150 Hz).

Digital PLLs. As explained in the last section, a digital PLL works with binary input signals. Phase detectors for a digital PLL are simpler than the fast multipliers used for linear PLLs; usual choices are XOR gates, edge-triggered JK flip-flops, or paired RS flip-flops with some additional gates [255, pp. 60–65]. The phase detector shown in Fig. 7.46 is the most complex, but it provides phase and frequency sensitivity and a quasi-infinite hold-in range.

Modified counters are used as loop filters for DPLLs. These may be $N/(N + K)$ counters or multistage counters, such as an N -by- M divider, where separate UP and DOWN counters are used, and a third counter measures the UP/DOWN difference. It is then possible to break off further signal processing if a certain threshold is not met. For the DCO, any typical all-digital frequency synthesizer, such as an accumulator, divider, or multiplicative generator may be used. The most frequently used synthesizer is the

**Fig. 7.47.** “Pulse-stealing” PLL [264]

tunable divider, popular because of its low phase error. The low resolution of this synthesizer can be improved by using a low receiver IF [263].

One DPLL realization with very low complexity is the 74LS297 circuit, which utilizes a “pulse-stealing” design. This scheme may be improved with the phase- and frequency-sensitive J-K flip-flop, as displayed in Fig. 7.47. The PLL works as follows: the “detect flip-flop” runs with rest frequency

Table 7.17. Hardware complexity of a pulse-stealing DPLL (4 CLBs \approx one Spartan-6 slice) [263]

Function group	CLBs
DCO	4
Phase detector	1.25
Loop filter	1.5
Averaging	2.75
PC-interface	2.5
Frequency synthesizer	6.5
State machine	2.5
Total	21

$$F_{\text{comp}} = \frac{F_{\text{in}}}{N} = \frac{F_{\text{osc}}}{KM}. \quad (7.37)$$

To allow tracking of incoming frequencies higher than the rest frequency, the oscillator frequency $F_{\text{osc}+}$ is set slightly higher:

$$T_{\text{comp}} - T_{\text{comp}+} = \frac{1}{2}T_{\text{osc}}, \quad (7.38)$$

such that the signal at point B oscillates half a period faster than the F_{osc} signal. After approximately two periods at the rest frequency, a one will be latched in the detector flip-flop. This signal runs through the deglitch and delay flip-flops, and then inhibits one pulse of the $\div K$ divider (thus the name “pulse-stealing”). This delays the signal at B such that the phase of signal A runs after B , and the cycle repeats. The lock-in range of the PLL has a lower bound of $F_{\text{in}}|_{\min}=0 \text{ Hz}$. The upper bound depends on the maximum output frequency $F_{\text{osc+}}/K$, so the lock-in range becomes

$$\pm \Delta\omega_L = \pm N \times F_{\text{osc+}}/(K \times M). \quad (7.39)$$

A receiver can be simplified by leaving out the counters N and M . In a WeatherFAX image-decoding application the second IF of the double-superhet receiver is set in such a way that the frequency modulation of 300 Hz ($\delta f = 0 \text{ Hz} \rightarrow \text{white}; \delta f = 300 \text{ Hz} \rightarrow \text{black}$) corresponds to exactly 32 steal pulses, so that the steal pulses correspond directly to the grayscale level. We set a pixel rate of 1920 Baud, and a IF of 16.6 kHz. For each pixel, four “steal values” (number of steal pulses in an interval) are determined, so a total of $\log_2(2 \times 4) = 3$ bit shifts are used to compute the 16 gray-level values. Table 7.17 shows the hardware complexity (21 CLBs \approx 14.5 Spartan-6 slices) of this PLL type.

Costas loop. This extended type of coherent loop was first proposed by John P. Costas in 1956, who used the loop for carrier recovery. As shown in Fig. 7.44, the CL has an *in-phase* and a *quadrature* path (subscripted I and Q there). With the $\pi/2$ phase shifter and the third PD and lowpass, the CL is approximately twice as complex as the PLL, but locks onto a signal twice as fast. Costas loops are very sensitive to small differences between in-phase and quadrature gain, and should therefore always be realized as all-digital circuits. The FPGA seems to be an ideal realization vehicle ([265, 266]).

For a signal $U(t) = A(t) \sin(\omega_0 t + \Delta\phi(t))$ we get, after the mixer and lowpass filters,

$$U_I(t) = K_d A(t) \cos(\Delta\phi(t)) \quad (7.40)$$

$$U_Q(t) = K_d A(t) \sin(\Delta\phi(t)), \quad (7.41)$$

where $2K_d$ is the gain of the PD. $U_I(t)$ and $U_Q(t)$ are then multiplied together in a third PD, and are lowpass filtered to get the DCO control signal:

$$U_{I \times Q}(t) \sim K_d \sin(2\Delta\phi(t)). \quad (7.42)$$

A comparison of (7.36) and (7.42) shows that, for small modulations of $\Delta\phi(t)$, the slope of the control signal $U_{I \times Q}(t)$ is twice the PLL’s. As in the PLL, if only FM or PM demodulation is needed, the PDs may be all digital.

Figure 7.48 shows a block diagram of a CL. The antenna signal is first filtered and amplified by a fourth order Butterworth bandpass, then digitized

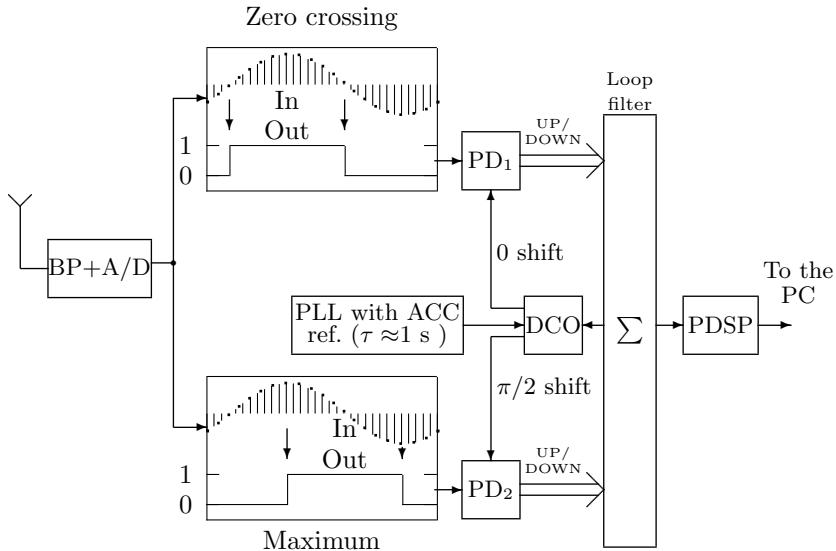


Fig. 7.48. Structure of the Costas loop

by an 8-bit converter at a sampling rate 32 or 64 times the carrier base frequency. The resulting signal is split, and fed into a zero-crossing detector and a minimum/maximum detector. Two phase detectors compare the signals with a reference signal, and its $\pi/2$ -shifted counterpart, synthesized by a high time-constant PLL with a reference accumulator [4, section 2]. Each phase detector has two edge detectors, which should generate a total of 4 UP and 4 DOWN signals. If more UP signals than DOWN are generated by the PDs, then the reference frequency is too low, and if more DOWN signals are generated, it is too high. The differences $\sum \text{UP} - \sum \text{DOWN}$ are accumulated for one pixel duration in a 13-bit accumulator acting as a loop filter. The loop filter data are passed to a pixel converter, which gives “correction values” to the DCO as shown in Table 7.18. The accumulated sums are also used as greyscale values for the pixel, and passed onto a PC to store and display the WeatherFAX pictures.

The smallest detectable phase offset for a 2 kBaud pixel rate is

$$f_{\text{carrier}+1} = \frac{1}{1/f_{\text{carrier}} - t_{ph37} \times 2 \text{ kBaud}/f_{\text{carrier}}} = 117.46 \text{ kHz}, \quad (7.43)$$

where $t_{ph37} = 1/(32 \times 117 \text{ kHz}) = 266 \text{ ns}$ is the sampling period at 32-times oversampling. The frequency resolution is $117.46 \text{ kHz} - (f_{\text{carrier}} = 117.4 \text{ kHz}) = 60 \text{ Hz}$. With a frequency modulation of 300 Hz, five greyscale values can be distinguished.

For a maximum phase offset of π , the loop will require a maximum locking time T_L of $\lceil 16/3 \rceil = 6$ samples, or about $1.5 \mu\text{s}$. Table 7.19 shows the complexity of the CL for 32 and 64 times oversampling.

Table 7.18. Loop filter output and DCO correction values at 32-times oversampling

Accumulator						
Under-flow	Over-flow	Sum	DCO-IN	gray value	$f_{\text{carrier}} \pm \delta f$	
Yes	No	$s < -(2^{13} - 1)$	3	0	+180 Hz	
No	No	$-(2^{13} - 1) \leq s < -2048$	2	0	+120 Hz	
No	No	$-2048 \leq s < -512$	1	4	+60 Hz	
No	No	$-512 \leq s < 512$	0	8	+0 Hz	
No	No	$512 \leq s < 2048$	-1	12	-60 Hz	
No	No	$2048 \leq s < 2^{13} - 1$	-2	15	-120 Hz	
No	Yes	$s \geq 2^{13} - 1$	-3	15	-180 Hz	

Table 7.19. Complexity of a fully digital Costas loop (4 CLBs \approx one Spartan-6 slice) [262, p. 60]

Function group	CLBs with oversampling	
	32 times	64 times
Frequency synthesizer	33	36
Zero detection	42	42
Maximum detection	16	16
Four phase detectors	8	8
Loop filter	51	51
DCO	12	15
TMS interface	11	11
Sum	173	179

Exercises

Note: If you have no prior experience with the Quartus II software, refer to the case study found in Sect. 1.4.3, p. 32. If not otherwise noted use the EP4CE115F29C7 from the Cyclone IV E family for the Quartus II synthesis evaluations.

7.1: The following MATLAB code can be used to compute the order of an element:

```
function N = order(x,M)
% Compute the order of x modulo M
p = x; l=1;
while p ~= 1
    l = l+1; p = p * x;
    re =real(p); im = imag(p);
    p = mod(re,M) + i * mod(im,M);
end;
N=1;
```

If, for instance, the function is called with `order(2,2^25+1)` the result is 50. To compute the single factors of $2^{25} + 1$, the standard MATLAB function `factor(2^25+1)` can be used.

For

- (a) $\alpha = 2$ and $M = 2^{41} + 1$
- (b) $\alpha = -2$ and $M = 2^{29} - 1$
- (c) $\alpha = 1 + j$ and $M = 2^{29} + 1$
- (d) $\alpha = 1 + j$ and $M = 2^{26} - 1$

compute the transform length, the “bad” factors ν (i.e., order not equal $\text{order}(\alpha, 2^B \pm 1)$), all “good” prime factors M/ν , and the available input bit width $B_x = (\log_2(M/\nu) - \log_2(L))/2$.

7.2: To compute the inverse $x^{-1} \bmod M$ for $\gcd(x, M) = 1$ of the value x , we can use the fact that the following diophantic equation holds:

$$\gcd(x, M) = u \times x + v \times M \quad \text{with } u, v \in \mathbb{Z}. \quad (7.44)$$

(a) Explain how to use the MATLAB function `[g u v]=gcd(x,M)` to compute the multiplication inverse.

Compute the following multiplicative inverses if possible:

- (b) $3^{-1} \bmod 73$;
- (c) $64^{-1} \bmod 2^{32} + 1$;
- (d) $31^{-1} \bmod 2^{31} - 1$;
- (e) $89^{-1} \bmod 2^{11} - 1$;
- (f) $641^{-1} \bmod 2^{32} + 1$.

7.3: The following MATLAB code can be used to compute Fermat NTTs for length 2, 4, 8, and 16 modulo 257:

```

function Y = ntt(x)
% Compute Fermat NTT of length 2,4,8 and 16 modulo 257
l = length(x);
switch (l)
    case 2, alpha=-1;
    case 4, alpha=16;
    case 8, alpha=4;
    case 16, alpha=2;
    otherwise, disp('NTT length not supported')
end
A=ones(1,l); A(2,2)=alpha;
%*****Computing second column
for m=3:l
    A(m,2)=mod(A(m-1,2)* alpha, 257);
end
%*****Computing rest of matrix
for m=2:l
    for n=2:l-1
        A(m,n+1)=mod(A(m,n)*A(m,2),257);
    end
end
%*****Computing NTT A*x
for k = 1:l
    C1 = 0;
    for i=1:l
        C1=C1+A(k,i);
    end
    A(k,:)=C1;
end
Y=A*x;

```

```

for j = 1:l
    C1 = C1 + A(k,j) * x(j);
end
X(k) = mod(C1, 257);
end
Y=X;

```

- (a) Compute the NTT \mathbf{X} of $\mathbf{x} = \{1, 1, 1, 1, 0, 0, 0, 0\}$.
(b) Write the code for the appropriate INTT. Compute the INTT of \mathbf{X} from part (a).
(c) Compute the element-by-element product $\mathbf{Y} = \mathbf{X} \odot \mathbf{X}$ and INTT(\mathbf{Y}) = \mathbf{y} .
(d) Extend the code for a complex Fermat NTT and INTT for $\alpha = 1 + j$. Test your program with the identity $\mathbf{x} = \text{INTT}(\text{NTT}(\mathbf{x}))$.

7.4: The Walsh transform for $N = 4$ is given by:

$$\mathbf{W}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix}.$$

- (a) Compute the scalar product of the row vectors. What property does the matrix have?
(b) Use the results from (a) to compute the inverse \mathbf{W}_4^{-1} .
(c) Compute the 8×8 Walsh matrix \mathbf{W}_8 , by scaling the original row vector by two (i.e., $h[n/2]$) and computing an additional two “children” $h[n] + h[n - 4]$ and $h[n] - h[n - 4]$ from row 3 and 4. There should be no zero in the resulting \mathbf{W}_8 matrix.
(d) Draw a function tree to construct Walsh matrices of higher order.

7.5: The Hadamard matrix can be computed using the following iteration:

$$\mathbf{H}_{2^{l+1}} = \begin{bmatrix} \mathbf{H}_{2^l} & \mathbf{H}_{2^l} \\ \mathbf{H}_{2^l} & -\mathbf{H}_{2^l} \end{bmatrix}, \quad (7.45)$$

with $\mathbf{H}_1 = [1]$.

- (a) Compute \mathbf{H}_2 , \mathbf{H}_4 , and \mathbf{H}_8 .
(b) Find the appropriate index for the rows in \mathbf{H}_4 and \mathbf{H}_8 , compared with the Walsh matrix \mathbf{W}_4 and \mathbf{W}_8 from Exercise 7.4.
(c) Determine the general rule to map a Walsh matrix into a Hadamard matrix.

Hint: First compute the index in binary notation.

7.6: The following MATLAB code can be used to compute the state-space description for $p_{14} = x^{14} + x^5 + x^3 + x^1 + 1$, the nonzero elements using `nnz`, and the maximum fan-in:

```

p= input('Please define power of matrix = ')
A=zeros(14,14);
for m=1:13
    A(m,m+1)=1;
end
A(14,14)=1;
A(14,13)=1;
A(14,11)=1;
A(14,9)=1;
Ap=mod(A^p,2);

```

```
nnz(Ap)
max(sum(Ap,2))
```

- (a) Compute the number of nonzero elements and fan-in for $p = 2$ to 8.
- (b) Modify the code to compute the twin $p_{14} = x^{14} + x^{13} + x^{11} + x^9 + 1$. Compute the number of nonzero elements for the modified polynomial for $p = 2$ to 8.
- (c) Modify the original code to compute the alternative LFSR implementation (see Fig. 7.18, p. 495) for (a) and (b) and compute the nonzero elements for $p = 2$ to 8.

7.7: (a) Compile the code for the length-6 LFSR **lfsr.vhd** from Example 7.1 (p. 495) using MODELSIM via **vcom -93 lfsr.vhd**

- (b) For the line

```
ff(1) <= NOT (ff(5) XOR ff(6));
```

substitute

```
ff(1) <= ff(5) XNOR ff(6);
```

and compile with ModelSim via **vcom -93 lfsr.vhd**.

- (c) Now compile again using MODELSIM VHDL-1987 interface via **vcom -87 lfsr.vhd**. Explain the results.

Note: Using Quartus II will not produce any differences by changing the VHDL version settings from VHDL 1993 to VHDL 1987.

8. Adaptive Systems

Many systems we have discussed so far had been designed for applications where the requirements for the “optimal” coefficients did not change over time. However, many real-world signals we find in typical DSP fields like speech processing, communications, radar, sonar, seismology, or biomedicine require that the “optimal” system coefficients need to be adjusted over time depending on the input signal. If the parameter changes slowly compared with the sampling frequency we can compute a “better” estimation for our optimal coefficients and adjust the system appropriately.

In general, any filter structure, FIR or IIR, with the many architectural variations we have discussed before, may be used as an adaptive digital filter (ADF). Comparing the different structural options, we note that:

- For FIR filters the direct form from Fig. 3.1 (p. 180) seems to be advantageous because the coefficient update can be done at the same time for all coefficients.
- For IIR filters the lattice structure shown in Fig. 4.12 (p. 237) seems to be a good choice because lattice filters possess a low fixed-point arithmetic roundoff error sensitivity and a simplified stability control of the coefficients.

From the published literature, however, it appears that FIR filters have been used more successfully than IIR filters and our focus in this chapter will therefore be efficient and fast implementation of adaptive FIR filters.

FIR filter algorithms should converge to the optimum nonrecursive estimator solution given (originally for continuous signal) through the Wiener–Hopf equation [267]. We will then discuss the optimum recursive estimator (Kalman filter). We will compare the different options in terms of computational complexity, stability of the algorithms, initial speed of convergence, consistency of convergence, and robustness to additive noise.

We then continue with more difficult tasks such as blind source separation (BSS) of multichannel signals using principle component analysis (PCA) and independent component analysis (ICA). Finally, speech processing algorithms are discussed ranging from ADPCM to MP3 format.

Adaptive systems can now be seen to be a mature DSP field. Many books were first published in the mid-1980s and can be used for a more in-depth study [268–273]. More recent results may be found in textbook like [274–

276]. Recent journal publications like IEEE Transactions on Signal Processing show, especially in the area of stability of LMS and its variations, essential research activity.

8.1 Application of Adaptive Systems

Although the application fields of adaptive filters are quite broad in nature, they can usually be described with one of the following four system configurations:

- Interference cancellation
- Prediction
- Inverse modeling
- Identification

We wish to discuss in the following the basic idea of these systems and present some typical successful applications for these classes. Although it may not always exactly describe the nature of the specific signals it is common to use the following notation for all systems, namely

x = input to the adaptive system

y = output of the adaptive system

d = desired response (of the adaptive system)

$e = d - y$ = estimation error

8.1.1 Interference Cancellation

In these very popular applications of the adaptive filter the incoming signal contains, beside the information-bearing signal, also an interference, which may, for example, be a random white noise or the 50/60 Hz power-line hum. Figure 8.1 shows the configuration for this application. The incoming (sensor) signal $d[n]$ and the adaptive filter output response $y[n]$ to a reference signal $x[n]$ is used to compute the error signal $e[n]$, which is also the system output in the interference cancellation configuration. Thus, after convergence, the (modified) reference signal, which will represent the additive inverse of the interference is subtracted from the incoming signal.

We will later study a detailed example of the interference cancellation of the power-line hum. A second popular application is the adaptive noise cancellation of echoes on telephone systems. Interference cancellation has also been used in an array of antennas (called beamformer) to adaptively remove noise interfering from unknown directions.

8.1.2 Prediction

In the prediction application the task of the adaptive filter is to provide a best prediction (usually in the least mean square sense) of a present value of a random signal. This is obviously only possible if the input signal is essential different from white noise. Prediction is illustrated in Fig. 8.2. It can be seen that the input $d[n]$ is applied over a delay to the adaptive filter input, as well as to compute the estimation error.

The predictive coding has been successfully used in image and speech signal processing. Instead of coding the signal directly, only the prediction error is encoded for transmission or storage. Other applications include the modeling of power spectra, data compression, spectrum enhancement, and event detection [269].

8.1.3 Inverse Modeling

In the inverse modeling structure the task is to provide an inverse model that represents the best fit (usually in the least squares sense) to an unknown time-varying plant. A typical communication example would be the task to estimate the multipath propagation of the signal to approximate an ideal transmission. The system shown in Fig. 8.3 illustrates this configuration. The input signal $d[n]$ enters the plant and the output of the unknown plant $x[n]$ is the input to the adaptive filter. A delayed version of the input $d[n]$ is then used to compute the error signal $e[n]$ and to adjust the filter coefficients of the adaptive filter. Thus, after convergence, the adaptive filter transfer

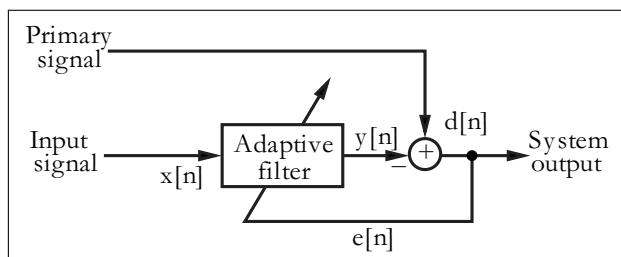


Fig. 8.1. Basic configuration for interference cancellation

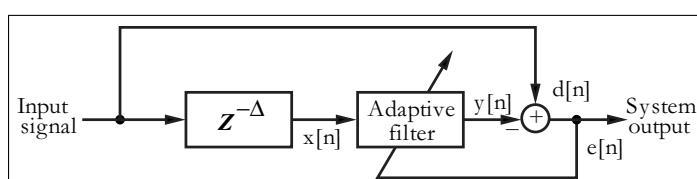


Fig. 8.2. Block diagram for prediction

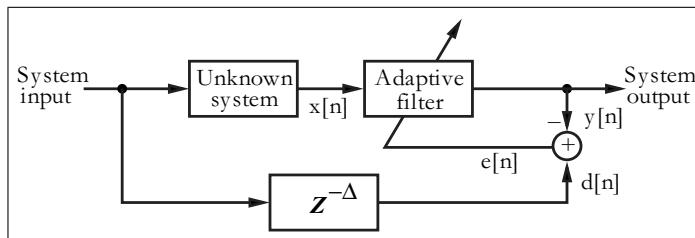


Fig. 8.3. Schematic diagram illustrating the inverse system modeling

function approximates the inverse of the transfer function of the unknown plant.

Besides the already-mentioned equalization in communication systems, inverse modeling with adaptive filters has been successfully used to improve S/N ratio for additive narrowband noise, for adaptive control systems, in speech signal analysis, for deconvolution, and digital filter design [269].

8.1.4 System Identification

In a system identification application the task is that the filter coefficients of the adaptive filter represent an unknown plant or filter. The system identification is shown in Fig. 8.4 and it can be seen that the time series, $x[n]$, is input simultaneously to the adaptive filter and another linear plant or filter with unknown transfer function. The output of the unknown plant $d[n]$ becomes the output of the entire system. After convergence the adaptive filter output $y[n]$ will approximate $d[n]$ in an optimum (usually least mean squares) sense. Provided that the order of the adaptive filter matches the order of the unknown plant and the input signal $x[n]$ is WSS the adaptive filter coefficients will converge to the same values as the unknown plant. In a practical application there will normally be an additive noise present at the output of the unknown plant (observation errors) and the filter structure will not exactly match that of the unknown plant. This will result in deviation from the perfect performance described. Due to the flexibility of this structure and the ability to individually adjust a number of input parameters independently it is one of the structures often used in the performance evaluations of adaptive filters. We will use these configurations to make a detailed comparison between LMS and RLS, the two most popular algorithms to adjust the filter coefficient of an adaptive filter.

Such system identification has been used for modeling in biology, or to model social and business systems, for adaptive control systems, digital filter design, and in geophysics [269]. In a seismology exploration, such systems have been used to generate a layered-earth model to unravel the complexities of the earth's surface [268].

8.2 Optimum Estimation Techniques

Signal properties. In order to use successfully the adaptive filter algorithms presented in the following and to guarantee the convergence and stability of the algorithms, it is necessary to make some basic assumptions about the nature of our input signals, which from a probabilistic standpoint, can be seen as a vector of random variables. First, the input signal (i.e., the random variable vector) should be *ergodic*, i.e., statistical properties like mean

$$\eta = E\{x\} = \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{n=0}^{N-1} x[n]$$

or variance

$$\sigma^2 = E\{(x - \eta)^2\} = \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{n=0}^{N-1} (x[n] - \eta)^2$$

computed using a single input signal should show the same statistical properties like the average over an ensemble of such random variables. Second, the signals need to be wide sense stationary (WSS), i.e., statistics measurements like average or variance measured over the ensemble averages are not a function of time, and the autocorrelation function

$$\begin{aligned} r[\tau] &= E\{x[t_1]x[t_2]\} = E\{x[t + \tau]x[t]\} \\ &= \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{n=0}^{N-1} x[n]x[n + \tau] \end{aligned}$$

depends only on the difference $\tau = t_1 - t_2$. We note in particular that

$$r[0] = E\{x[t]x[t]\} = E\{|x[t]|^2\} \quad (8.1)$$

computes the average power of the WSS process.

Some of the more advanced adaptive algorithms require knowledge of higher moments than first or second order. The most often needed is the

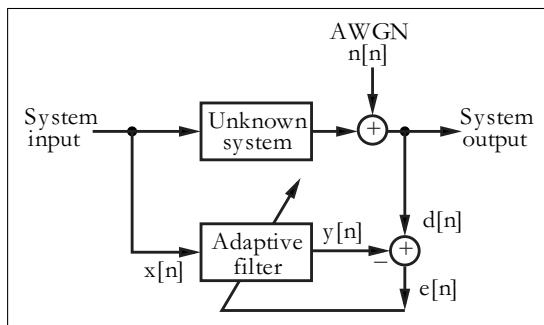


Fig. 8.4. Basic configuration for system identification

fourth order statistic called *kurtosis*. In the literature we find several different definitions and we will use the (normalized without -3) version supported by MATLAB, i.e.,

$$\kappa = \frac{E\{(x - \eta)^4\}}{(E\{(x - \eta)^2\})^2} \quad (8.2)$$

This definition in MATLAB has the nice property that any DC offset or scale factor in the signal does not change the kurtosis value, i.e.,

$$\kappa(ax + b) = \kappa(x) \quad (8.3)$$

Let us calculate the kurtosis, as an example, for a DC free uniform distribution within the bounds $[-a, a]$. The probability density function (PDF) is constant at $1/(2a)$. We need first to determine the fourth and second moments. With $\eta = 0$ it follows that:

$$\begin{aligned} E\{x^4\} &= \frac{1}{2a} \int_{-a}^a x^4 dx = \frac{1}{2a} \left. \frac{x^5}{5} \right|_{-a}^a = \frac{1}{2a} 2 \frac{a^5}{5} = \frac{a^4}{5} \\ E\{x^2\} &= \frac{1}{2a} \int_{-a}^a x^2 dx = \frac{1}{2a} \left. \frac{x^3}{3} \right|_{-a}^a = \frac{1}{2a} 2 \frac{a^3}{3} = \frac{a^2}{3} \end{aligned}$$

and finally for the kurtosis we get

$$\kappa = \frac{E\{(x - \eta)^4\}}{(E\{(x - \eta)^2\})^2} = \frac{a^4/5}{(a^2/3)^2} = \frac{9}{5} = 1.8 \quad (8.4)$$

One prominent kurtosis is based on the Gaussian or normal distribution. The kurtosis for the Gaussian distribution is 3 when using MATLAB. Distributions that have a kurtosis higher than Gaussian are natural signals, have an impulse type distribution, and are called *supergaussian* or *leptokurtic*. A signal for instance of length L with I impulses and otherwise zero has $\kappa = L/I$. Most technical signals we measure such as sine, cosine, triangle, or binary ± 1 “coin toss” signals have a kurtosis less than the Gaussian. These signals are characterized as *subgaussian* or *platykurtic*. Table 8.1 gives an overview of the mean, variance, and kurtosis of some typical PDFs.

Definition of cost function. The definition of the cost function applied to the estimator output is a critical parameter in all adaptive filter algorithms. We need to “weight” somehow the estimation error

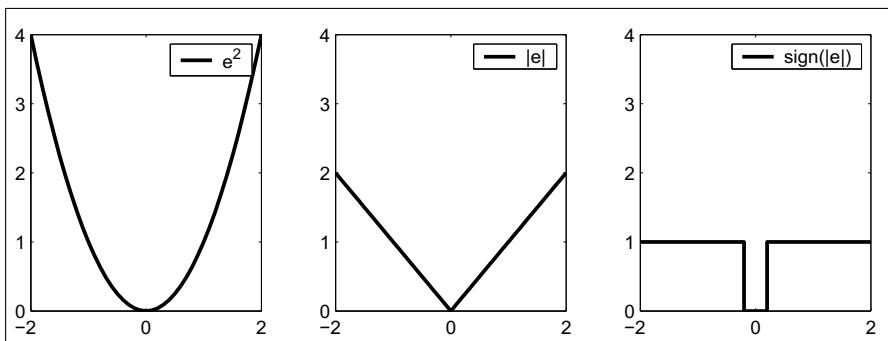
$$e[n] = d[n] - y[n], \quad (8.5)$$

where $d[n]$ is the random variable to be estimated, and $y[n]$ is the computed estimate via the adaptive filter. The most commonly used cost function is the least-mean-squares (LMS) function given as

$$J = E\{e^2[n]\} = \overline{(d[n] - y[n])^2}. \quad (8.6)$$

Table 8.1. Mean, variance, and kurtosis of example distributions

Distribution	Mean	Variance	Kurtosis
Coin toss ± 1	0.0	1.0	1.0
sine/cosine period > 8	0.0	0.5	1.5
triangle $[0,10]$	5.0	35.0	1.8
Uniform $[-0.5, 0.5]$	0.0	1/12	1.8
$0, 1, -1, 0, 1, -1, \dots$	0.0	0.5	2.0
Gaussian/normal	0.0	1.0	3.0
Rayleigh noise	1.25	0.43	3.3
Laplace noise	0.0	2.0	5.8
Flute	0.0	1.0	3.59
Piano	0.0	1.0	3.94
Male speech	0.0	1.0	16.4
Female speech	0.0	1.0	46.1
I Impulse/length L	0.0	1.0	L/I

**Fig. 8.5.** Three possible error cost functions

It should be noted that this is not the only cost function that may be used. Alternatives are functions such as the absolute error or the nonlinear threshold functions as shown in Fig 8.5 on the right. The nonlinear threshold type may be used if a certain error level is acceptable and, as we will see later, it can reduce the computational burden of the adaptation algorithm. It may be interesting to note that the original adaptive filter algorithms by Widrow [271] use such a threshold function for the error.

On the other hand, the quadratic error function of the LMS method will enable us to build a stochastic gradient approach based on the Wiener–Hopf relation originally developed in the continuous signal domain. We review the Wiener–Hopf estimation in the next section, which will lead directly to the popular LMS adaptive filter algorithms first proposed by Widrow et al. [277, 278].

8.2.1 The Optimum Wiener Estimation

The output of the adaptive FIR filter is computed via the convolution sum

$$y[n] = \sum_{k=0}^{L-1} f_k x[n-k], \quad (8.7)$$

where the filter coefficients f_k have to be adjusted in such a way that the defined cost function J is minimum. It is, in general, more convenient to write the convolution with vector notations according to

$$y[n] = \mathbf{x}^T[n] \mathbf{f} = \mathbf{f}^T \mathbf{x}[n], \quad (8.8)$$

with $\mathbf{f} = [f_0 f_1 \dots f_{L-1}]^T$, $\mathbf{x}[n] = [x[n] x[n-1] \dots x[n-(L-1)]]^T$, are size $(L \times 1)$ vectors and T means matrix transposition or the Hermitian transposition for complex data. For $\mathbf{A} = [a[k, l]]$ the transposed matrix is “mirrored” at the main diagonal, i.e., $\mathbf{A}^T = [a[l, k]]$. Using the definition of the error function (8.5) we get

$$e[n] = d[n] - y[n] = d[n] - \mathbf{f}^T \mathbf{x}[n]. \quad (8.9)$$

The mean square error function now becomes

$$\begin{aligned} J &= E\{e^2[n]\} = E\{d[n] - y[n]\}^2 = E\{d[n] - \mathbf{f}^T \mathbf{x}[n]\}^2 \\ &= E\{(d[n] - \mathbf{f}^T \mathbf{x}[n])(d[n] - \mathbf{f}^T \mathbf{x}[n])\} \\ &= E\{d[n]^2 - 2d[n]\mathbf{f}^T \mathbf{x}[n] + \mathbf{f}^T \mathbf{x}[n]\mathbf{x}^T[n]\mathbf{f}\}. \end{aligned} \quad (8.10)$$

Note that the error is a quadratic function of the filter coefficients that can be pictured as a concave hyperparaboloidal surface, a function that never goes negative; see Fig. 8.6 for an example with two filter coefficients. Adjusting the filter weights to minimize the error involves descending along this surface with the objective of getting to the bottom of the bowl. Gradient methods are commonly used for this purpose. The choice of mean square type of cost function will enable a well-behaved quadratic error surface with a single unique minimum. The cost is minimum if we differentiate (8.10) with respect to \mathbf{f} and set this gradient to zero, i.e.,

$$\nabla = \frac{\partial J}{\partial \mathbf{f}^T} = E\{(-2d[n]\mathbf{x}[n] + 2\mathbf{x}[n]\mathbf{x}^T[n]\mathbf{f}_{\text{opt}})\} = 0.$$

Assuming that the filter weight vector \mathbf{f} and the signal vector $\mathbf{x}[n]$ are statistically independent (i.e., uncorrelated), it follows, that:

$$E\{d[n]\mathbf{x}[n]\} = E\{\mathbf{x}[n]\mathbf{x}^T[n]\}\mathbf{f}_{\text{opt}},$$

then the optimal filter coefficient vector \mathbf{f}_{opt} can be computed with,

$$\mathbf{f}_{\text{opt}} = E\{\mathbf{x}[n]\mathbf{x}^T[n]\}^{-1} E\{d[n]\mathbf{x}[n]\}. \quad (8.11)$$

The expectation terms are usually defined as follows:

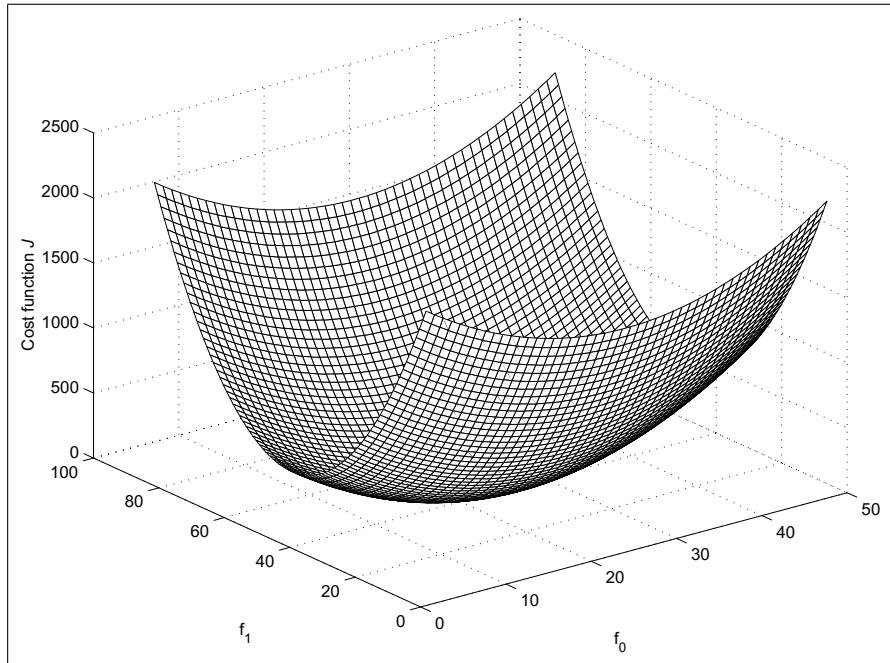


Fig. 8.6. Error cost function for the two-component case. The minimum of the cost function is at $f_0 = 25$ and $f_1 = 43.3$

$$\begin{aligned}
 \mathbf{R}_{xx} &= E\{\mathbf{x}[n]\mathbf{x}^T[n]\} \\
 &= E \left[\begin{array}{ccc} x[n]x[n] & x[n]x[n-1] & \dots x[n]x[n-(L-1)] \\ x[n-1]x[n] & x[n-1]x[n-1] & \dots \\ \vdots & \ddots & \vdots \\ x[n-(L-1)]x[n] & \dots & \end{array} \right] \\
 &= \left[\begin{array}{ccc} r[0] & r[1] & \dots r[L-1] \\ r[1] & r[0] & \dots r[L-2] \\ \vdots & \ddots & \vdots \\ r[L-1] & r[L-2] & \dots r[0] \end{array} \right]
 \end{aligned}$$

is the $(L \times L)$ autocorrelation matrix of the input signal sequence, which has the form of the Toeplitz matrix, and

$$\begin{aligned}
 \mathbf{r}_{d\mathbf{x}} &= E\{d[n]\mathbf{x}[n]\} \\
 &= E \left[\begin{array}{c} d[n]x[n] \\ d[n]x[n-1] \\ \vdots \\ d[n]x[n-(L-1)] \end{array} \right] = \left[\begin{array}{c} r_{dx}[0] \\ r_{dx}[1] \\ \vdots \\ r_{dx}[L-1] \end{array} \right]
 \end{aligned}$$

is the $(L \times 1)$ cross-correlation vector between the desired signal and the reference signal. With these definitions we can now rewrite (8.11) more compactly as

$$\mathbf{f}_{\text{opt}} = \mathbf{R}_{\mathbf{x}\mathbf{x}}^{-1} \mathbf{r}_{d\mathbf{x}}. \quad (8.12)$$

This is commonly recognized as the Wiener–Hopf equation [267], which yields the optimum LMS solution for the filter coefficient vector \mathbf{f}_{opt} . One requirement to have a unique solution for (8.12) is that $\mathbf{R}_{\mathbf{x}\mathbf{x}}^{-1}$ exist, i.e., the autocorrelation matrix must be nonsingular, or put differently, the determinate is nonzero. Fortunately, it can be shown that for WSS signals the $\mathbf{R}_{\mathbf{x}\mathbf{x}}$ matrix is nonsingular [268, p. 41] and the inverse exists.

Using (8.10) the residue error of the optimal estimation becomes:

$$\begin{aligned} \mathbf{J}_{\text{opt}} &= E\{d[n] - \mathbf{f}_{\text{opt}}^T \mathbf{x}[n]\}^2 \\ &= E\{d[n]\}^2 - 2\mathbf{f}_{\text{opt}}^T \mathbf{r}_{d\mathbf{x}} + \mathbf{f}_{\text{opt}}^T \underbrace{\mathbf{R}_{\mathbf{x}\mathbf{x}} \mathbf{f}_{\text{opt}}}_{\mathbf{r}_{d\mathbf{x}}} \\ \mathbf{J}_{\text{opt}} &= \mathbf{r}_{dd}[0] - \mathbf{f}_{\text{opt}}^T \mathbf{r}_{d\mathbf{x}}, \end{aligned} \quad (8.13)$$

where $\mathbf{r}_{dd}[0] = \sigma_d^2$ is the variance of d .

We now wish to demonstrate the Wiener–Hopf algorithm with the following example.

Example 8.1: Two-Tap FIR Filter Interference Cancellation

Suppose we have an observed communication signal that consists of three components: The information-bearing signal, which is a Manchester encoded sensor signal $m[n]$ with amplitude $B = 10$, shown in Fig. 8.7a; an additive white Gaussian noise $n[n]$, shown in Fig. 8.7b; and a 60-Hz power-line hum interference with amplitude $A = 50$, shown in Fig. 8.7c. Assuming the sampling frequency is 4 times the power-line hum frequency, i.e., $4 \times 60 = 240$ Hz, the observed signal can therefore be formulated as follows:

$$d[n] = A \cos[\pi n/2] + Bm[n] + \sigma^2 n[n].$$

The reference signal $x[n]$ (shown in Fig. 8.7d), which is applied to the adaptive filter input, is given as

$$x[n] = \cos[\pi n/2 + \phi],$$

where $\phi = \pi/6$ is a constant offset. The two-tap filter then has the following output:

$$y[n] = f_0 \cos\left[\frac{\pi}{2}n + \phi\right] + f_1 \cos\left[\frac{\pi}{2}(n-1) + \phi\right].$$

To solve (8.12) we compute first the autocorrelation for $x[n]$ with delays 0 and 1:

$$r_{xx}[0] = E\{(\cos[\pi n/2 + \phi])^2\} = \frac{1}{2}$$

$$r_{xx}[1] = E\{\cos[\pi n/2 + \phi] \sin[\pi(n-1)/2 + \phi]\} = 0.$$

For the cross-correlation we get

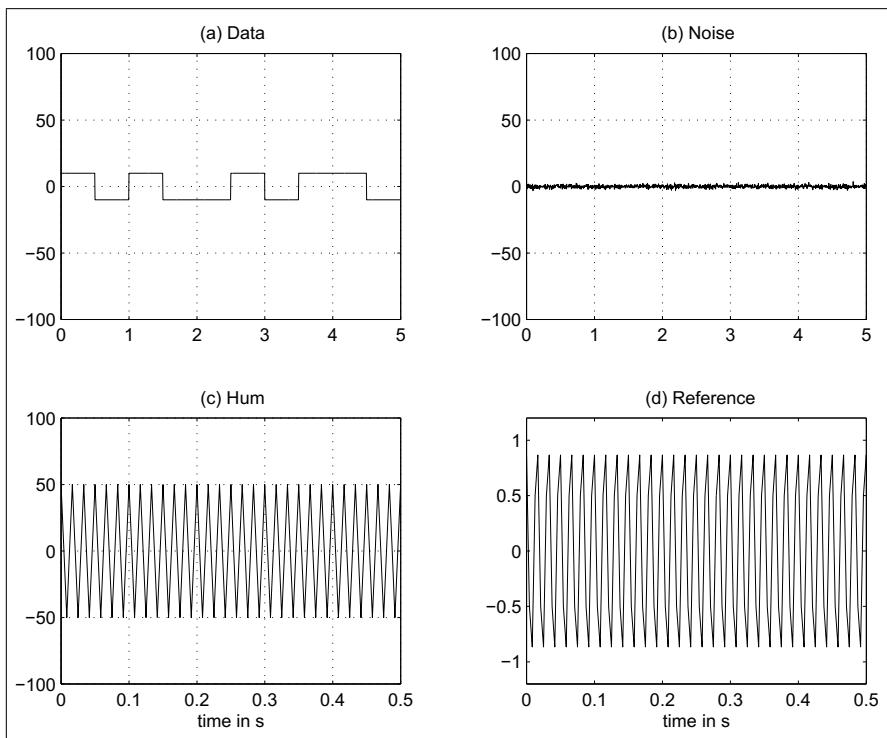


Fig. 8.7. Signals used in power-line hum Example 8.1

$$\begin{aligned}
 r_{dx}[0] &= E \left\{ (A \cos[\pi n/2] + Bm[n] + \sigma^2 n[n]) \cos[\pi n/2 + \phi] \right\} \\
 &= \frac{A}{2} \cos(\phi) = 12.5\sqrt{3} \\
 r_{dx}[1] &= E \left\{ (A \cos[\pi n/2] + Bm[n] + \sigma^2 n[n]) \sin[\pi n/2 + \phi] \right\} \\
 &= \frac{A}{2} \cos(\phi - \pi) = \frac{50}{4} = 12.5.
 \end{aligned}$$

As required for the Wiener–Hopf equation (8.12) we can now compute the (2×2) autocorrelation matrix and the (2×1) cross-correlation vector and get

$$\begin{aligned}
 \mathbf{f}_{\text{opt}} &= \mathbf{R}_{xx}^{-1} \mathbf{r}_{dx} \begin{bmatrix} r_{xx}[0] & r_{xx}[1] \\ r_{xx}[1] & r_{xx}[0] \end{bmatrix}^{-1} \begin{bmatrix} r_{dx}[0] \\ r_{dx}[1] \end{bmatrix} \\
 &= \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}^{-1} \begin{bmatrix} 12.5\sqrt{3} \\ 12.5 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 12.5\sqrt{3} \\ 12.5 \end{bmatrix}
 \end{aligned}$$

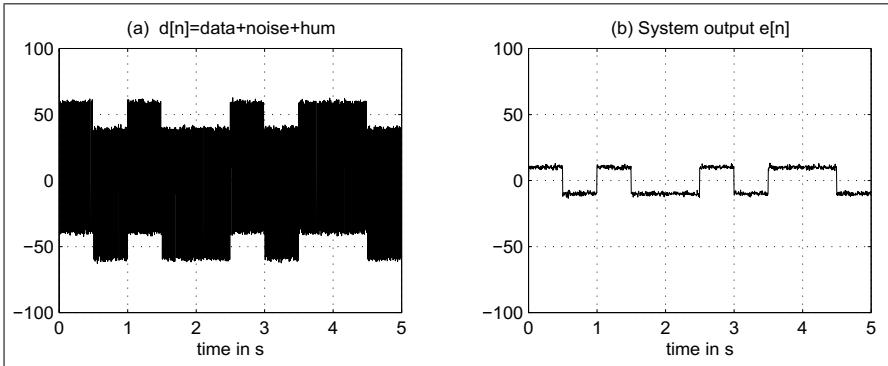


Fig. 8.8. Canceling 60-Hz power-line interference of a Manchester-coded data signal using optimum Wiener estimation

$$= \begin{bmatrix} 25\sqrt{3} \\ 25 \end{bmatrix} = \begin{bmatrix} 43.3 \\ 25 \end{bmatrix}.$$

The simulation of these data is shown in Fig. 8.8. It shows (a) the sum of the three signals (Manchester-coded 5 bits, power-line hum of 60 Hz, and the additive white Gaussian noise) and the system output (i.e., $e[n]$) with the canceled power-line hum.

8.1

8.3 The Widrow–Hoff Least Mean Square Algorithm

There may exist a couple of reasons why we wish to avoid a direct computation of the Wiener estimation (8.12). First, the generation of the autocorrelation matrix \mathbf{R}_{xx} and the cross-correlation vector \mathbf{r}_{dx} are already computationally intensive. We need to compute the autocorrelation of \mathbf{x} and the cross-correlation between \mathbf{d} and \mathbf{x} and we may, for instance, not know how many data samples we need to use in order to have sufficient statistics. Secondly, if we have constructed the correlation functions we still have to compute the inverse of the autocorrelation matrix \mathbf{R}_{xx}^{-1} , which can be very time consuming, if the filter order gets larger. Even if a procedure is available to invert \mathbf{R}_{xx} , the precision of the result may not be sufficient because of the many computational steps involved, especially with a fixed-point arithmetic implementation.

The Widrow–Hoff least mean square (LMS) adaptive algorithm [277] is a practical method for finding a close approximation to (8.12) in real time. The algorithm does not require explicit measurement of the correlation functions, nor does it involve matrix inversion. Accuracy is limited by statistical sample size, since the filter coefficient values are based on the real-time measurements of the input signals.

The LMS algorithm is an implementation of the method of the steepest descent. According to this method, the next filter coefficient vector $\mathbf{f}[n+1]$ is equal to the present filter coefficient vector $\mathbf{f}[n]$ plus a change proportional to the negative gradient:

$$\mathbf{f}[n+1] = \mathbf{f}[n] - \frac{\mu}{2} \nabla[n]. \quad (8.14)$$

The parameter μ is the learning factor or step size that controls stability and the rate of convergence of the algorithm. During each iteration the true gradient is represented by $\nabla[n]$.

The LMS algorithm estimates an instantaneous gradient in a crude but efficient manner by assuming that the gradient of $J = e[n]^2$ is an estimate of the gradient of the mean-square error $E\{e[n]^2\}$. The relationship between the true gradient $\nabla[n]$ and the estimated gradients $\hat{\nabla}[n]$ is given by the following expression:

$$\nabla[n] = \left[\frac{\partial E\{e[n]^2\}}{\partial f_0}, \frac{\partial E\{e[n]^2\}}{\partial f_1}, \dots, \frac{\partial E\{e[n]^2\}}{\partial f_{L-1}} \right]^T \quad (8.15)$$

$$\begin{aligned} \hat{\nabla}[n] &= \left[\frac{\partial e[n]^2}{\partial f_0}, \frac{\partial e[n]^2}{\partial f_1}, \dots, \frac{\partial e[n]^2}{\partial f_{L-1}} \right]^T \\ &= 2e[n] \left[\frac{\partial e[n]}{\partial f_0}, \frac{\partial e[n]}{\partial f_1}, \dots, \frac{\partial e[n]}{\partial f_{L-1}} \right]^T. \end{aligned} \quad (8.16)$$

The estimated gradient components are related to the partial derivatives of the instantaneous error with respect to the filter coefficients, which can be obtained by differentiating (8.9), it follows that:

$$\hat{\nabla}[n] = -2e[n] \frac{\partial e[n]}{\partial \mathbf{f}} = -2e[n] \mathbf{x}[n]. \quad (8.17)$$

Using this estimate in place of the true gradient in (8.14) yields:

$$\mathbf{f}[n+1] = \mathbf{f}[n] - \frac{\mu}{2} \hat{\nabla}[n] = \mathbf{f}[n] + \mu e[n] \mathbf{x}[n]. \quad (8.18)$$

Let us summarize all necessary step for the LMS algorithm² in the following

² Note that in the original presentation of the algorithm [277] the update equation $\mathbf{f}[n+1] = \mathbf{f}[n] + 2\mu e[n] \mathbf{x}[n]$ is used because the differentiation of the gradient in (8.17) produces a factor 2. The update equation (8.18) follows the notation that is used in most of the current textbooks on adaptive filters.

Algorithm 8.2: Widrow–Hoff LMS Algorithm

The Widrow–Hoff LMS algorithm to adjust the L filter coefficients of an adaptive filter uses the following steps:

- 1) Initialize the $(L \times 1)$ vector $\mathbf{f} = \mathbf{x} = \mathbf{0} = [0, 0, \dots, 0]^T$.
- 2) Accept a new pair of input samples $\{x[n], d[n]\}$ and shift $x[n]$ in the reference signal vector $\mathbf{x}[n]$.
- 3) Compute the output signal of the FIR filter, via

$$y[n] = \mathbf{f}^T[n] \mathbf{x}[n]. \quad (8.19)$$

- 4) Compute the error function with

$$e[n] = d[n] - y[n]. \quad (8.20)$$

- 5) Update the filter coefficients according to

$$\mathbf{f}[n+1] = \mathbf{f}[n] + \mu e[n] \mathbf{x}[n]. \quad (8.21)$$

Now continue with step 2.

Although the LMS algorithm makes use of gradients of mean-square error functions, it does not require squaring, averaging, or differentiation. The algorithm is simple and generally easy to implement in software (MATLAB code see, for instance, [276, p. 332]; C code [279], or PDSP assembler code [280]).

A simulation using the same system configuration as in Example 8.1 (p. 542) is shown in Fig. 8.9 for different values of the step size μ . Adaptation starts after 1 second. System output $e[n]$ is shown in the left column and the filter coefficient adaptation on the right. We note that depending on the value μ the optimal filter coefficients approach $f_0 = 43.3$ and $f_1 = 25$.

It has been shown that the gradient estimate used in the LMS algorithm is unbiased and that the expected value of the weight vector converges to the Wiener weight vector (8.12) when the input signals are WSS, which was anyway required in order to be able to compute the inverse of the autocorrelation matrix $\mathbf{R}_{\mathbf{xx}}^{-1}$ for the Wiener estimate. Starting with an arbitrary initial filter coefficient vector, the algorithm will converge in the mean and will remain stable as long as the learning parameter μ is greater than 0 but less than an upper bound μ_{\max} . Figure 8.10 shows an alternative form to represent the convergence of the filter coefficient adaptation by a projection of the coefficient values in a (f_0, f_1) mapping. The figure also shows the contour line with equal error. It can be seen that the LMS algorithm moves in a zigzag way towards the minimum rather than the true gradient, which would move exactly orthogonal to these error contour lines.

Although the LMS algorithm is considerably simpler than the RLS algorithm (we will discuss this later) the convergence properties of the LMS algorithm are nonetheless difficult to analysis rigorously. The simplest approach to determine an upper bound of μ makes use of the eigenvalues of $\mathbf{R}_{\mathbf{xx}}$ by solving the homogeneous equation

$$\mathbf{0} = \det(\lambda \mathbf{I} - \mathbf{R}_{\mathbf{xx}}), \quad (8.22)$$

where \mathbf{I} is the $L \times L$ identity matrix. There are L eigenvalues $\lambda[k]$ that have the following properties:

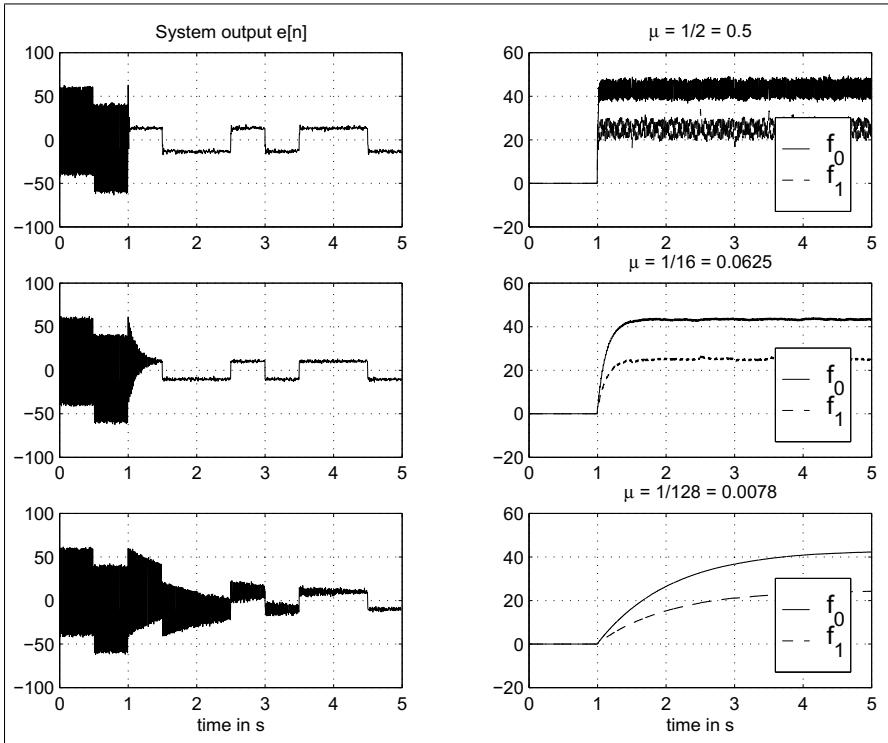


Fig. 8.9. Simulation of the power-line interference cancellation using the LMS algorithm for three different values of the step size μ . System output $e[n]$; (left) Filter coefficients (right)

$$\det(\mathbf{R}_{xx}) = \prod_{k=0}^{L-1} \lambda[k] \quad \text{and} \quad (8.23)$$

$$\text{trace}(\mathbf{R}_{xx}) = \sum_{k=0}^{L-1} \lambda[k]. \quad (8.24)$$

From this eigenvalue analysis of the autocorrelation matrix it follows that the LMS algorithm will be stable (in the mean sense) if

$$0 < \mu < \frac{2}{\lambda_{\max}}. \quad (8.25)$$

Although the filter is assumed to be stable, we will see later that this upper bound will not guarantee a finite mean square error, i.e., that $\mathbf{f}[n]$ converges to \mathbf{f}_{opt} and a much more stringent bound has to be used.

The simulation of the LMS algorithm in Fig. 8.9 also reveals the underlying exponential nature of the individual learning curves. Using the eigenvalue

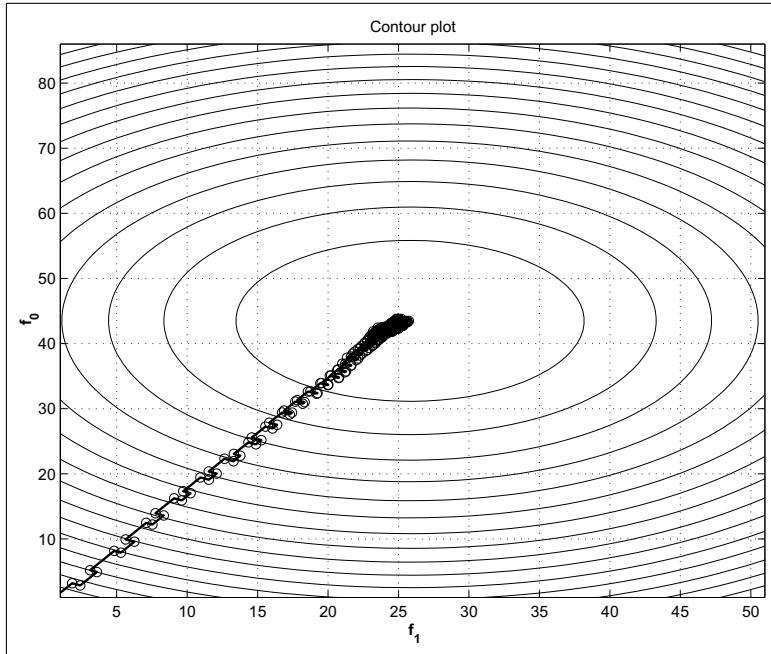


Fig. 8.10. Demonstration of the convergence of the power-line interference example using a 2D contour plot for $\mu = 1/16$

analysis we may also transform the filter coefficient in independent so-called “modes” that are no longer linear dependent. The number of natural modes is equal to the number of degrees of freedom, i.e., the number of independent components and in our case identically with the number of filter coefficients. The time constant of the k^{th} mode is related to the k eigenvalue $\lambda[k]$ and the parameter μ by

$$\tau[k] = \frac{1}{2\mu\lambda[k]}. \quad (8.26)$$

Hence the longest time constant, τ_{\max} , is associated with the smallest eigenvalue, λ_{\min} via

$$\tau_{\max} = \frac{1}{2\mu\lambda_{\min}}. \quad (8.27)$$

Combining (8.25) and (8.27) gives

$$\tau_{\max} > \frac{\lambda_{\max}}{2\lambda_{\min}}, \quad (8.28)$$

which suggests that the larger the eigenvalue ratio (EVR), $\lambda_{\max}/\lambda_{\min}$ of the autocorrelation matrix \mathbf{R}_{xx} the longer the LMS algorithm will take to converge. Simulation results that confirm this finding can be found for instance, in [275, p. 64] and will be discussed, in Sect. 8.3.1 (p. 551).

The results presented so far on the ADF stability can be found in most original published work by Widrow and many textbooks. However, these conditions do *not* guarantee a finite variance for the filter coefficient vector, neither do they guarantee a finite mean-square error! Hence, as many users of the algorithm realized, considerably more stringent conditions are required to ensure convergence of the algorithm. In the examples in [276, p. 130], for instance, you find the “rule of thumb” that a factor 10 smaller values for μ should be used.

More recent results indicate that the bound from (8.25) must be more restrictive. For example, the results presented by Horowitz and Senne [281] and derived in a different way by Feuer and Weinstein [282] show that the step size (assuming that the elements of the input vector $\mathbf{x}[n]$ are statistically independent) has to be restricted via the two conditions:

$$0 < \mu < \frac{1}{\lambda_l} \quad l = 0, 1, \dots, L - 1 \quad \text{and} \quad (8.29)$$

$$\sum_{l=0}^{L-1} \frac{\mu \lambda_l}{1 - \mu \lambda_l} < 2, \quad (8.30)$$

to ensure convergence. These conditions cannot be solved analytically, but it can be shown that they are closely bounded by the following condition:

$$0 < \mu < \frac{2}{3 \times \text{trace}(\mathbf{R}_{xx})} = \frac{2}{3 \times L \times r_{xx}[0]}. \quad (8.31)$$

The upper bound of (8.31) has a distinct practical advantage. Trace of \mathbf{R}_{xx} is, by definition, (see (8.24), p. 547) the total average input signal power of the reference signal, which can easily be estimated from the reference signal $x[n]$.

Example 8.3: Bounds on Step Size

From the analysis in (8.25) we see that we first need to compute the eigenvalues of the \mathbf{R}_{xx} matrix, i.e.

$$\mathbf{0} = \det(\lambda \mathbf{I} - \mathbf{R}_{xx}) = \begin{bmatrix} r_{xx}[0] - \lambda & r_{xx}[1] \\ r_{xx}[1] & r_{xx}[0] - \lambda \end{bmatrix} \quad (8.32)$$

$$= \det \begin{bmatrix} 0.5 - \lambda & 0 \\ 0 & 0.5 - \lambda \end{bmatrix} = (0.5 - \lambda)^2 \quad (8.33)$$

$$\lambda[1, 2] = 0.5. \quad (8.34)$$

Using (8.25) gives

$$\mu_{\max} = \frac{2}{\lambda_{\max}} = 4. \quad (8.35)$$

Using the more restrictive bound from (8.31) yields

$$\mu_{\max} = \frac{2}{L \times 3 \times r_{xx}[0]} = \frac{2}{3 \times 2 \times 0.5} = \frac{2}{3}. \quad (8.36)$$

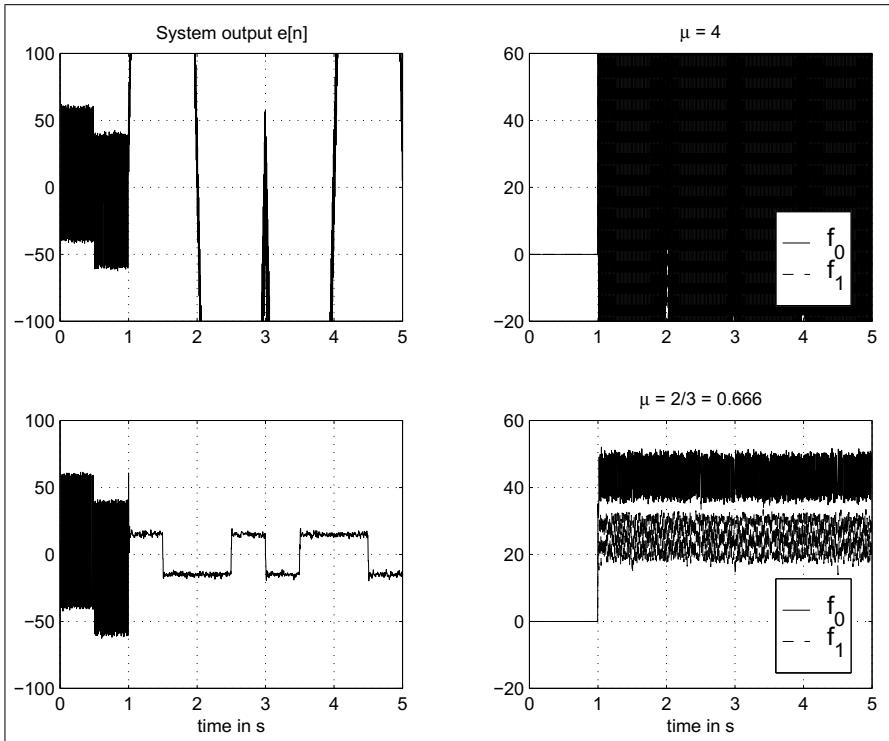


Fig. 8.11. Simulation of the power-line interference cancellation using the maximum step size values for the LMS algorithm. System output $e[n]$ (left); filter coefficients (right)

The simulation results in Fig. 8.11 indicate that in fact $\mu = 4$ does not show convergence, while $\mu = 2/3$ converges. 8.3

We also note from the simulation shown in Fig. 8.11 that even with $\mu_{\max} = 2/3$ the convergence is much faster, but the coefficients “ripple around” essentially. Much smaller values for μ are necessary to have a smooth approach of the filter coefficient to the optimal values and to stay there.

The condition found by Horowitz and Senne [281] and Feuer and Weinstein [282] made the assumption that all inputs $x[n]$ are statistically independent. This assumption is true if the input data come, for instance, from an antenna array of L independent sensors, however, for ADFs with the tapped delay structure, it has been shown, for instance, by Butterweck [268], that for a long filter the stability bound can be relaxed to

$$0 < \mu < \frac{2}{L \times r_{xx}[0]}, \quad (8.37)$$

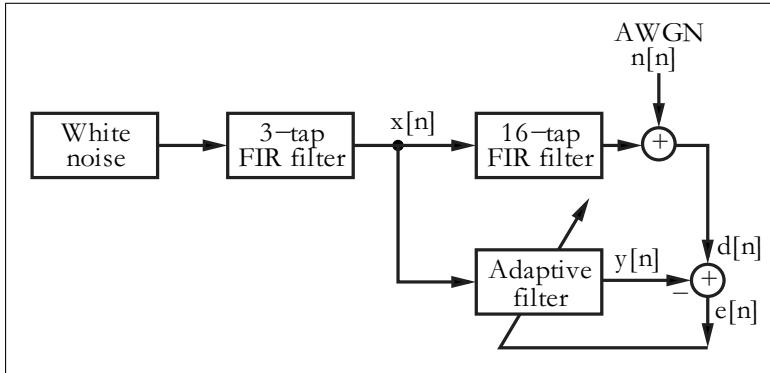


Fig. 8.12. System identification configuration for LMS learning curves

i.e., compared with (8.31) the upper bound can be relaxed by a factor of 3 in the denominator. But the condition (8.37) only applies for a long filter and it may therefore be better to use (8.31).

8.3.1 Learning Curves

Learning curve, i.e., the error function J displayed over the number of iterations is an important measurement instrument when comparing the performance of different algorithms and system configurations. We wish in the following to study the LMS algorithm regarding the eigenvalue ratio $\lambda_{\max}/\lambda_{\min}$ and the sensitivity to signal-to-noise (S/N) ratio in the system to be identified.

A typical performance measurement of adaptive algorithms using a system-identification problem is displayed in Fig. 8.12. The adaptive filter has a length of $L = 16$ the same length as the “unknown” system, whose coefficients have to be learned. The additive noise level behind the “unknown” system has been set to two different levels -10 dB for a high-noise environment and to -48 dB for a low-noise environment equivalent to an 8-bit quantization.

For the LMS algorithm the eigenvalue ratio (EVR) is the critical parameter that determines the convergence speed, see (8.28), p. 548. In order to generate a different eigenvalue ratio we use a white Gaussian noise source with $\sigma^2 = 1$ that is shaped by a digital filter. We may, for instance, use a first order IIR filter that generates a first order Markov process; see Exercise 8.10 (p. 628). We may alternatively filter the white noise by a three-tap symmetrical FIR filter whose coefficients are $c^T = [a, b, a]$. The FIR filter has the advantage that we can easily normalize the power. The coefficients should be normalized $\sum_k c_k^2 = 1$ in such a way that input and output sequences have the same power. This requires that

$$1 = a^2 + b^2 + a^2 \quad \text{or} \quad a = 0.5 \times \sqrt{1 - b^2}. \quad (8.38)$$

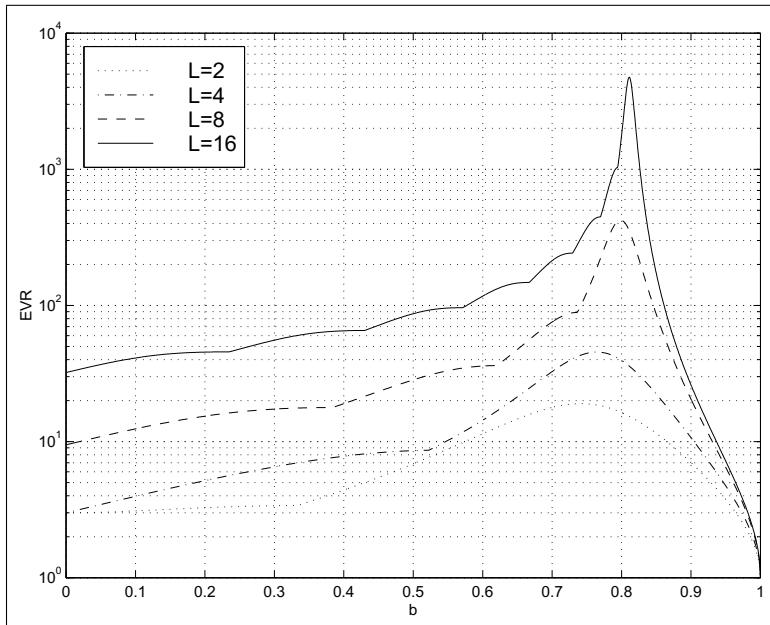


Fig. 8.13. Eigenvalues ratio for a three-tap filter system for different system size L

Table 8.2. Four different noise-shaping FIR filters to generate power-of-ten eigenvalue ratios for $L = 16$

No.	Impulse response	EVR
1	$0 + 1z^{-1} + 0.0z^{-2}$	1
2	$0.247665 + 0.936656z^{-1} + 0.247665z^{-2}$	10
3	$0.577582 + 0.576887z^{-1} + 0.577582z^{-2}$	100
4	$0.432663 + 0.790952z^{-1} + 0.432663z^{-2}$	1000

With this filter it is possible to generate different eigenvalue ratios $\lambda_{\max}/\lambda_{\min}$ as shown in Fig. 8.13 for different system size $L = 2, 4, 8$, and 16 . We can now use Table 8.2 to get power-of-ten EVRs for the system of length $L = 16$.

For a white Gaussian source the \mathbf{R}_{xx} matrix is a diagonal matrix $\sigma^2 \mathbf{I}$ and the eigenvalues are therefore all one, i.e., $\lambda_l = 1; l = 0, 1, \dots, L - 1$. The other EVRs can be verified with MATLAB, see Exercise 8.9 (p. 628). The impulse response of the unknown system g_k is an odd filter with coefficients $1, -2, 3, -4, \dots, -3, 2, -1$ as shown in Fig. 8.14a. The step size for the LMS algorithm has been determined with

$$\mu_{\max} = \frac{2}{3 \times L \times E\{\mathbf{x}^2\}} = \frac{1}{24}. \quad (8.39)$$

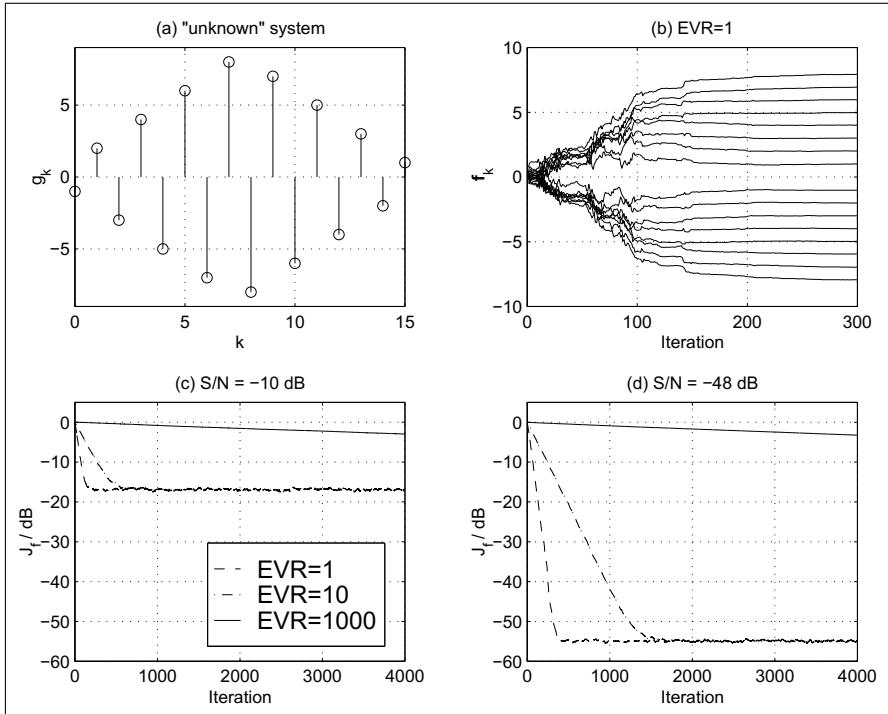


Fig. 8.14. Learning curves for the LMS algorithm using the system identification configuration shown in Fig. 8.12. (a) Impulse response g_k of the “unknown” system. (b) Coefficient learning over time. (c) Average over 50 learning curves for large system noise. (d) Average over 50 learning curves for small system noise

In order to guarantee perfect stability the step size has been chosen to be $\mu = \mu_{\max}/2 = 1/48$. The learning curve, or coefficient error is computed via the normalized error function

$$J[n] = 20 \log_{10} \left(\frac{\sum_{k=0}^{15} (g_k - f_k[n])^2}{\sum_{k=0}^{15} g_k^2} \right). \quad (8.40)$$

The coefficient adaptation for a single adaptation run with EVR=1 is shown in Fig. 8.14b. It can be seen that after 200 iterations the adaptive filter has learned the coefficient of the unknown system without an error. From the learning curves (average over 50 adaptation cycles) shown in Fig. 8.14c and d it can be seen that the LMS algorithm is very sensitive to the EVR. Many iterations are necessary in the case of the high EVR. Unfortunately, many real-world signals have high EVR. Speech signals, for instance, may have EVR of 1874 [283]. On the other hand, we see from Fig. 8.14c that the LMS algorithm still adapts well in a high-noise environment.

8.3.2 Normalized LMS (NLMS)

The LMS algorithm discussed so far uses a constant step size μ proportional to the stability bound $\mu_{\max} = 2/(L \times r_{xx}[0])$. Obviously this requires knowledge of the signal statistic, i.e., $r_{xx}[0]$, and this statistic must not change over time. It is, however, possible that this statistic changes over time, and we wish to adjust μ accordingly. These can be accomplished by computing a temporary estimate for the signal power via

$$r_{xx}[0] = \frac{1}{L} \mathbf{x}^T[n] \mathbf{x}[n], \quad (8.41)$$

and the “normalized” μ is given by

$$\mu_{\max}[n] = \frac{2}{\mathbf{x}^T[n] \mathbf{x}[n]}. \quad (8.42)$$

If we are concerned that the denominator can temporary become very small and μ too large, we may add a small constant δ to $\mathbf{x}^T[n] \mathbf{x}[n]$, which yields

$$\mu_{\max}[n] = \frac{2}{\delta + \mathbf{x}^T[n] \mathbf{x}[n]}. \quad (8.43)$$

To be on the safe side, we would not choose $\mu_{\max}[n]$. Instead we would use a somewhat smaller value, like $0.5 \times \mu_{\max}[n]$. The following example should demonstrate the normalized LMS algorithm.

Example 8.4: Normalized LMS

Suppose we have again the system identification configuration from Fig. 8.12 (p. 551), only this time the input signal $x[n]$ to the adaptive filter and the “unknown system” is the noisy pulse-amplitude-modulated (PAM) signal shown in Fig. 8.15a. For the conventional LMS we compute first $r_{xx}[0]$, and calculate $\mu_{\max} = 0.0118$. The step size for the normalized LMS algorithm is adjusted depending on the momentary power $\sum x[n]^2$ of the reference signal. For the computation of $\mu_{NLMS}[n]$ shown in Fig. 8.15c it can be seen that at times when the absolute value of the reference signal is large the step size is reduced and for small absolute values of the reference signal, a larger step size is used. The adaptation of the coefficient displayed over time in Fig. 8.15b reflects this issue. Larger learning steps can be seen at those times when $\mu_{NLMS}[n]$ is larger. An average over 50 adaptations is shown in the learning curves in Fig. 8.15d. Although the EVR of the noisy PAM is larger than 600, it can be seen that the normalized LMS has a positive effect on the convergence behavior of the algorithm.

8.4

The power estimation using (8.41) is a precise power snapshot of the current data vector $\mathbf{x}[n]$. It may, however, be desired to have a longer memory in the power computation to avoid a temporary small value and a large μ value. This can be accomplished using a recursive update of the previous estimations of the power, with

$$P[n] = \beta P[n-1] + (1 - \beta) |x[n]|^2, \quad (8.44)$$

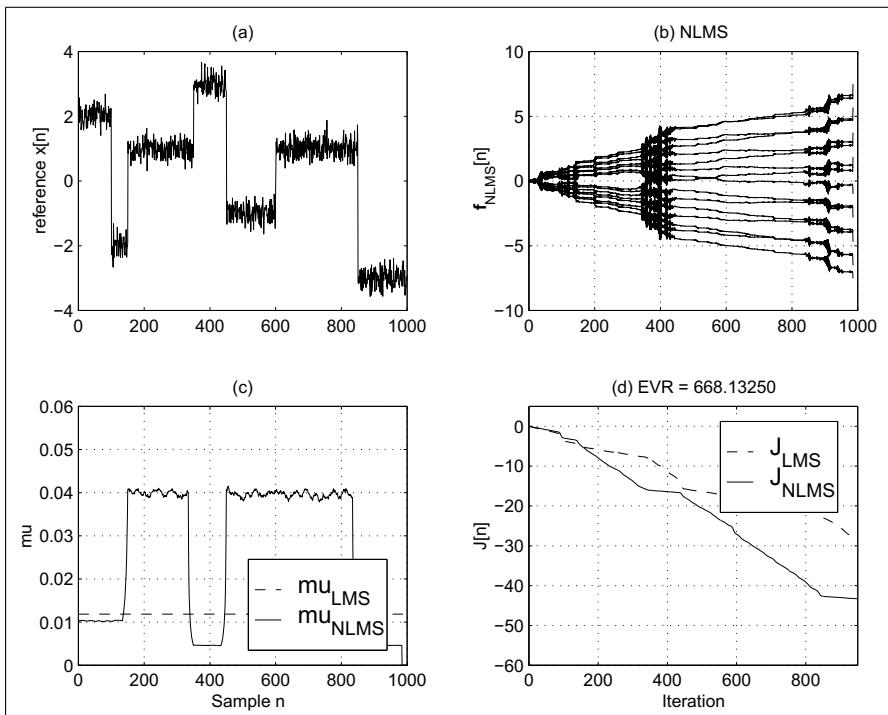


Fig. 8.15. Learning curves for the normalized LMS algorithm using the system identification configuration shown in Fig. 8.12. **(a)** The reference signal input $x[n]$ to the adaptive filter and the “unknown” system. **(b)** Coefficient learning over time for the normalized LMS. **(c)** Step size μ used for LMS and NLMS. **(d)** Average over 50 learning curves

with β less than but close to 1. For a nonstationary signal such as the one shown in Fig. 8.15 the choice of the parameter β must be done carefully. If we select β too small the NLMS will more and more have the performance of the original LMS algorithm; see Exercise 8.14 (p. 629).

8.4 Transform Domain LMS Algorithms

LMS algorithms that solve the filter coefficient adjustment in a transform domain have been proposed for two reasons. The goal of the *fast convolution techniques* [284] is to lower the computational effort, by using block update and transforming the convolution to compute the adaptive filter output and the filter coefficient adjustment in the transform domain with the help of a fast cyclic convolution algorithm. The second method that uses transform-domain techniques has the main goal to improve the adaptation rate of the

LMS algorithm, because it is possible to find transforms that allow a “decoupling” of the modes of the adaptive filter [283, 285].

8.4.1 Fast-Convolution Techniques

Fast cyclic convolution using transforms like FFTs can be applied to FIR filters. For the adaptive filter this leads to a block-oriented processing of the data. Although we may use any block size, the block size is usually chosen to be twice the size of the adaptive filter length so that the time delay in the coefficient update becomes not too large. It is also most often from a computational effort a good choice. In the first step a block of $2L$ input values $x[n]$ are convolved via transform with the filter coefficients \mathbf{f}_L , which produces L new filter output values $y[n]$. These results are then used to compute L error signals $e[n]$. The filter coefficient update is then done also in the transform domain, using the already transformed input sequence $x[n]$. Let us go through these block processing steps using a $L = 3$ example. We compute the three filter output signals in one block:

$$\begin{aligned}y[n] &= f_0x[n] + f_1[n]x[n - 1] + f_2[n]x[n - 2] \\y[n + 1] &= f_0x[n + 1] + f_1[n]x[n] + f_2[n]x[n - 1] \\y[n + 2] &= f_0x[n + 2] + f_1[n]x[n + 1] + f_2[n]x[n].\end{aligned}$$

These can be interpreted as a cyclic convolution of

$$\{f_0, f_1, f_2, 0, 0, 0\} \circledast \{x[n + 2], x[n + 1], x[n], x[n - 1], x[n - 2], 0\}.$$

The error signals follow then with:

$$\begin{aligned}e[n] &= d[n] - y[n] & e[n + 1] &= d[n + 1] - y[n + 1] \\e[n + 2] &= d[n + 2] - y[n + 2].\end{aligned}$$

The block processing for the filter gradient ∇ can now be written as

$$\begin{aligned}\nabla[n] &= e[n]\mathbf{x}[n] & \nabla[n + 1] &= e[n + 1]\mathbf{x}[n + 1] \\\nabla[n + 2] &= e[n + 2]\mathbf{x}[n + 2].\end{aligned}$$

The update for each individual coefficient is then computed with

$$\begin{aligned}\nabla_0 &= e[n]x[n] + e[n + 1]x[n + 1] + e[n + 2]x[n + 2] \\&\nabla_1 = e[n]x[n - 1] + e[n + 1]x[n] + e[n + 2]x[n - 1] \\&\nabla_2 = e[n]x[n - 2] + e[n + 1]x[n + 1] + e[n + 2]x[n].\end{aligned}$$

We again see that this is a cyclic convolution, only this time the input sequence $x[n]$ appears in reverse order

$$\begin{aligned}&\{0, 0, 0, e[n], e[n + 1], e[n + 2]\} \\&\circledast \{0, x[n - 2], x[n - 1], x[n], x[n + 1], x[n + 2]\}.\end{aligned}$$

In the Fourier domain the reverse order in time yields that we need to compute the conjugate transform of X . The coefficient update then becomes

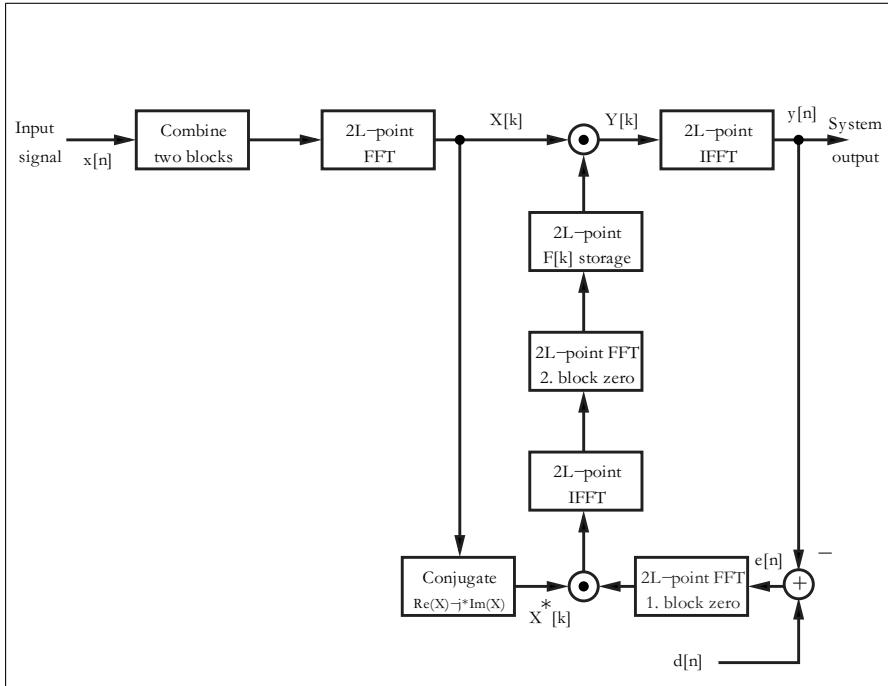


Fig. 8.16. Fast transform-domain filtering method using the FFT

$$\mathbf{f}[n+L] = \mathbf{f}[n] + \frac{\mu_B}{L} \nabla[n]. \quad (8.45)$$

Figure 8.16 shows all the necessary steps, when using the FFT for the fast convolution.

From the stability standpoint the block delay in the coefficient is not uncritical. Feuer [286] has shown that the step size has to be reduced to

$$0 < \mu_B < \frac{2B}{(B+2) \times \text{trace}(\mathbf{R}_{xx})} = \frac{2}{(1+2/B)L \times r_{xx}[0]}. \quad (8.46)$$

for a block update of B steps each. If we compare this result with the result for μ_{\max} from (8.31) page 549 we notice that the values are very similar. Only for large block sizes $B \gg L$ will the change in μ_B have considerable impact. This reduces to (8.31) for a block size of $B = 1$. However, the time constant is measured in blocks of L data and it follows that the largest time constant for the BLMS algorithm is L times larger than the largest time constant associated with the LMS algorithm.

8.4.2 Using Orthogonal Transforms

We have seen in Sect. 8.3.1 (p. 551) that the LMS algorithm is highly sensitive to the eigenvalue ratio (EVR). Unfortunately, many real-world signals

have high EVRs. Speech signals, for instance, may have EVR of 1874 [283]. But it is also well known that the transform-domain algorithms allow a “de-coupling” of the mode of the signals. The Karhunen–Loéve transform (KLT) is the optimal method in this respect, but unfortunately not a real time option; see Exercise 8.11 (p. 628). Discrete cosine transforms (DCT) and fast Fourier transform (FFT), followed by other orthogonal transforms like Walsh, Hadamard, or Haar are the next best choice in terms of convergence speed; see Exercise 8.13, (p. 628) [287, 288].

Let us try in the following to use this concept to improve the learning rate of the identification experiment presented in Sect. 8.3.1 (p. 551), where the adaptive filter has to “learn” an impulse response of an unknown 16-tap FIR filter, as shown in Fig. 8.12 (p. 551). In order to apply the transform techniques and still to monitor the learning progress we need to compute in addition to the LMS algorithm 8.2 (p. 546) the DCT of the incoming reference signal $\mathbf{x}[n]$ as well as the IDCT of the coefficient vector \mathbf{f}_n . In a practical application we do not need to compute the IDCT, it is only necessary to compute it once after we reach convergence. The following MATLAB code demonstrates the transform-domain DCT-LMS algorithm:

```

for k = L:Iterations
    x = [xin;x(1:L-1)]; % adapt over full length
    x = g'*x + n(k); % get new sample
    z = dct(x); % "unknown" filter output + AWGN
    z = dct(x);
    y = f' * z; % LxL orthogonal transform
    err = din-y; % transformed filter output
    err = din-y;
    f = f + err*mu.*z; % error: primary - reference
    f = f + err*mu.*z; % update weight vector
    fi = idct(f); % filter in original domain
    J(k-L+1) = J(k-L+1) + sum((fi-g).^2); % Learning curve
end

```

The effect of a transform \mathbf{T} on the eigenvalue spread can be computed via

$$\mathbf{R}_{zz} = \mathbf{T} \mathbf{R}_{xx} \mathbf{T}^H, \quad (8.47)$$

where the superscript H denotes the transpose conjugate.

The only thing we have not considered so far is that the L “modes” or frequencies of the transformed input signal $z[l]$ are now more or less statistically independent input vectors and the step size μ in the original domain may no longer be appropriate to guarantee stability, or allow fast convergence. In fact, the simulations by Lee and Un [288] show that if no power normalization is used in the transform domain then the convergence did not improve compared with the time-domain LMS algorithm. It is therefore reasonable to compute for these L spectral components different step sizes according to the stability bound (8.31), p. 549, just using the power of the transform components:

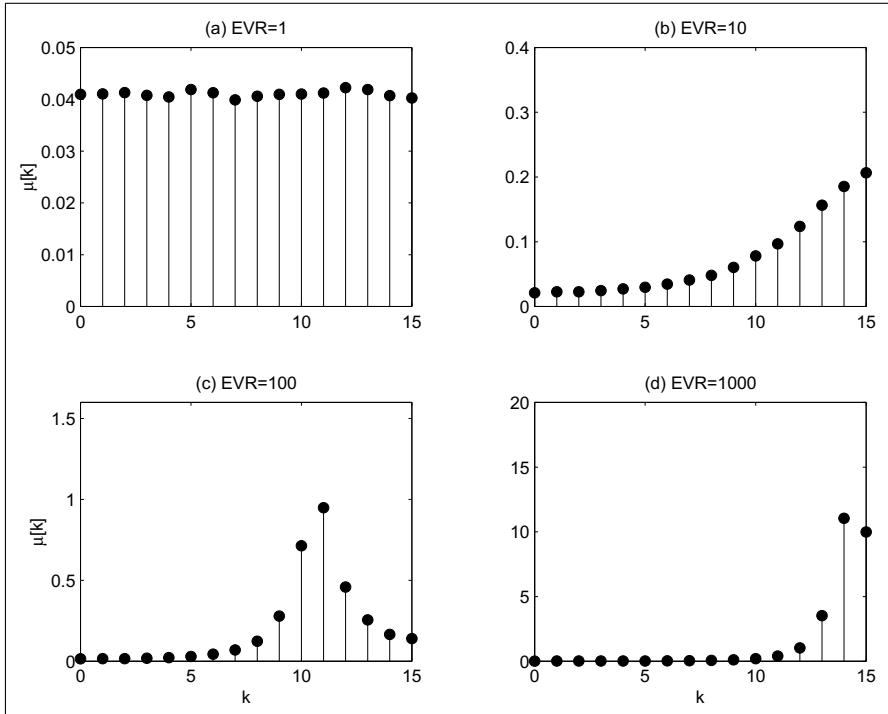


Fig. 8.17. Optimal step size for the DCT-LMS transform-domain algorithm using the system identification configuration shown in Fig. 8.12 (p. 551) for four different eigenvalue ratios. (a) Eigenvalue ratios of 1. (b) Eigenvalue ratios of 10. (c) Eigenvalue ratios of 100. (d) Eigenvalue ratios of 1000

$$\mu_{\max}[k] = \frac{2}{3 \times L \times r_{zz,k}[0]} \quad \text{for } k = 0, 1, \dots, L - 1.$$

The additional effort is now the computation of the power normalization of all L spectral components. The MATLAB code above already includes a componentwise update via $\mathbf{mu}.*\mathbf{z}$, where the $.*$ stands for the componentwise multiplication.

The adjustment in μ is somewhat similar to the normalized LMS algorithm we have discussed before. We may therefore use directly the power normalization update similar to (8.42) p. 554 for the frequency component. The effect of power normalization and transform \mathbf{T} on the eigenvalue spread can be computed via

$$\mathbf{R}_{zz} = \lambda^{-1} \mathbf{T} \mathbf{R}_{xx} \mathbf{T}^H \lambda^{-1}, \quad (8.48)$$

where λ^{-1} is a diagonal matrix that normalizes \mathbf{R}_{zz} in such a way that the diagonal elements all become 1, see [287].

Figure 8.17 shows the computed step sizes for four different eigenvalue ratios of the $L = 16$ FIR filter. For a pure Gaussian input all spectral com-

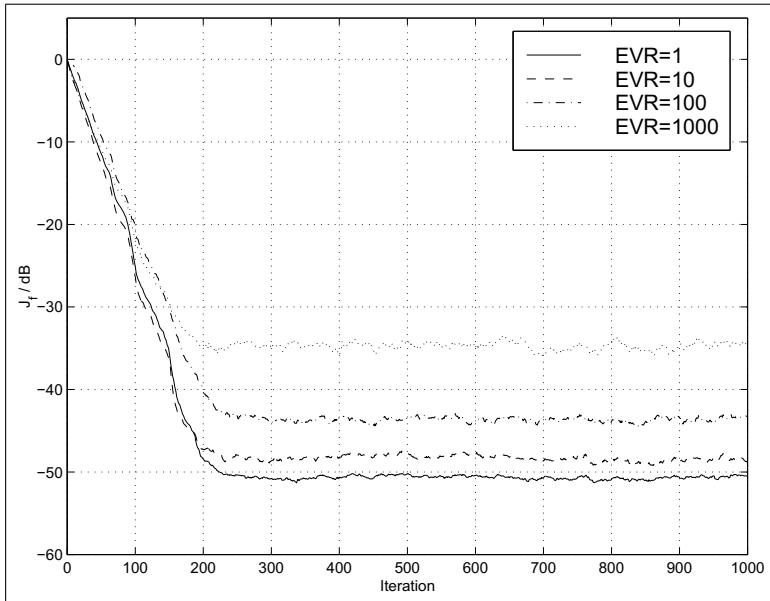


Fig. 8.18. Learning curves for the DCT transform-domain LMS algorithm using the system identification configuration shown in Fig. 8.12 (p. 551) for an average of 50 cycles using four different eigenvalue ratios

ponents should be equal and the step size is almost the same, as can be seen from Fig. 8.17a. The other filter shapes the noise in such a way that the power of these spectral components is increased (decreased) and the step size has to be set to a lower (higher) value.

From Fig. 8.18 the positive effect on the performance of the DCT-LMS transform-domain approach can be seen. The learning converges, even for very high eigenvalue ratios like 1000. Only the error floor and consistency of the error at -48 dB is not reached as well for high EVRs as for the lower EVRs.

One factor that must be considered in choosing the transform for real-time application algorithms is the computational complexity. In this respect, real transforms like DCT or DST transforms are superior to complex transform like the FFT, transforms with fast algorithms are better than the algorithms without. Integer transforms like Haar or Hadamard, that do not need multiplications at all, are desirable [287]. Lastly, we also need to take into account that the RLS (discussed later) is another alternative, which has, in general, a higher complexity than the LMS algorithm, but may be more efficient than a transform-domain filter approach and also yield as fast a convergence as the KLT-based LMS algorithm.

8.5 Implementation of the LMS Algorithm

We now wish to look at the task to implement the LMS algorithm with FPGAs. Before we can proceed with a HDL design, however, we need to ensure that quantization effects are tolerable. Later in this section we will then try to improve the throughput by using pipelining, and we need to ensure then also that the ADF is still stable.

8.5.1 Quantization Effects

Before we can start to implement the LMS algorithm in hardware we need to ensure that the parameter and data are well in the “green” range. This can be done if we change the software simulation from full precision to the desired integer precision. Figure 8.19 shows the simulation for 8-bit integer data and $\mu = 1/4, 1/8$ and $1/16$. Note that we cannot choose μ too small, otherwise we will no longer get convergence through the large scaling of the gradient $e[n]x[n]$ with μ in the coefficient update equation (8.21), p. 546. The smaller the step size μ the more problem the algorithm has to converge to the optimal values $f_0 = 43.3$ and $f_1 = 25$. This is somehow a contrary requirement to the upper bound on μ given through the stability requirement of the algorithm. It can therefore be necessary to add fractional bits to the system to overcome these two contradictions.

8.5.2 FPGA Design of the LMS Algorithm

A possible implementation of the algorithm represented as a signal flow graph is shown in Fig. 8.20. From a hardware implementation standpoint we note that we need one scaling for μ and $2L$ general multipliers. The effort is therefore more than twice the effort of the programmable FIR filter as discussed in Chap. 3, Example 3.1 (p. 182).

We wish to study in the following the FPLD implementation of the LMS algorithm.

Example 8.5: Two-Tap Adaptive LMS FIR Filter

The VHDL design³ for a filter with two coefficients f_0 and f_1 with a step size of $\mu = 1/4$ is shown in the following listing:

```
-- This is a generic LMS FIR filter generator
-- It uses W1 bit data/coefficients bits

LIBRARY ieee; USE ieee.std_logic_1164.ALL;
PACKAGE n_bit_int IS                         -- User defined types
    CONSTANT W1 : INTEGER := 8;   -- Input bit width
    CONSTANT W2 : INTEGER := 16; -- Multiplier bit width 2*W1
    SUBTYPE SLV1 IS STD_LOGIC_VECTOR(W1-1 DOWNTO 0);
```

³ The equivalent Verilog code `fir_lms.v` for this example can be found in Appendix A on page 861. Synthesis results are shown in Appendix B on page 881.

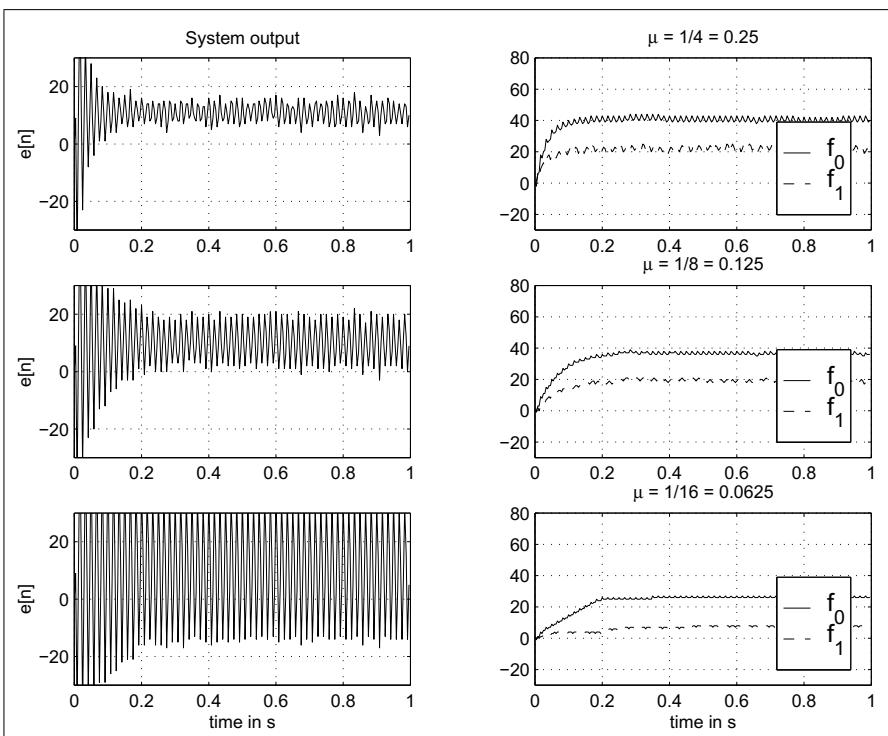


Fig. 8.19. Simulation of the power-line interference cancellation using the LMS algorithm for integer data. (left) System output $e[n]$. (right) filter coefficients

```

SUBTYPE SLV2 IS STD_LOGIC_VECTOR(W2-1 DOWNTO 0);
END n_bit_int;
LIBRARY work; USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-----
ENTITY fir_lms IS
    GENERIC (L : INTEGER := 2);      -----> Interface
    PORT ( clk   : IN STD_LOGIC;    -- Filter length
           reset : IN STD_LOGIC;  -- System clock
           x_in  : IN SLV1;      -- Asynchronous reset
           d_in  : IN SLV1;      -- System input
           f0_out: OUT SLV1;    -- Reference input
           f1_out: OUT SLV1;    -- 1. filter coefficient
           y_out : OUT SLV2;    -- 2. filter coefficient
           e_out : OUT SLV2);   -- System output
END fir_lms;

```

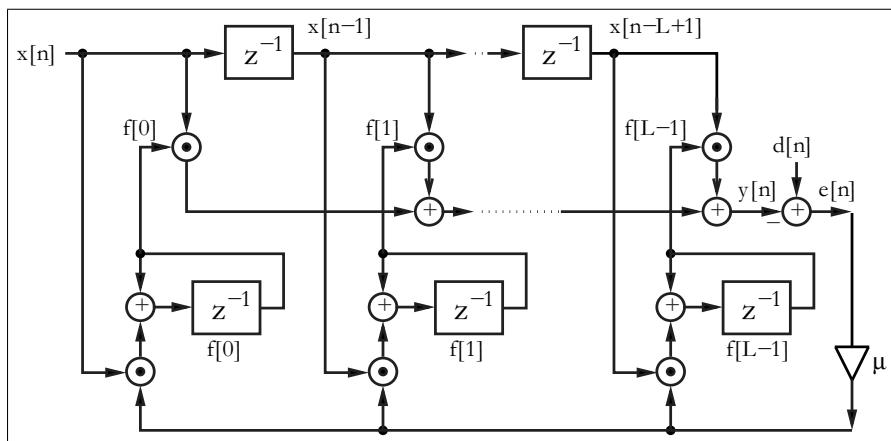


Fig. 8.20. Signal flow graph of the LMS algorithm

```

-----  

ARCHITECTURE fpga OF fir_lms IS  

TYPE AO_L1SLV1 IS ARRAY (0 TO L-1) OF SLV1;  

TYPE AO_L1SLV2 IS ARRAY (0 TO L-1) OF SLV2;  

SIGNAL d          : SLV1;  

SIGNAL emu       : SLV1;  

SIGNAL y, sxty   : SLV2;  

SIGNAL e, sxtb   : SLV2;  

SIGNAL x, f      : AO_L1SLV1; -- Coeff/Data arrays  

SIGNAL p, xemu  : AO_L1SLV2; -- Product arrays  

BEGIN  

dsxt: PROCESS (d) -- 16 bit signed extension for input d
BEGIN
    sxtb(7 DOWNTO 0) <= d;
    FOR k IN 15 DOWNTO 8 LOOP
        sxtb(k) <= d(d'high);
    END LOOP;
END PROCESS;  

Store: PROCESS(clk, reset) --> Store data or coefficients
BEGIN
    IF reset = '1' THEN                      -- Asynchronous clear
        d <= (OTHERS => '0');
        x(0) <= (OTHERS => '0'); x(1) <= (OTHERS => '0');
        f(0) <= (OTHERS => '0'); f(1) <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
        d <= d_in;
        x(0) <= x_in;
    END IF;
END PROCESS;

```

```

        x(1) <= x(0);
        f(0) <= f(0) + xemu(0)(15 DOWNTO 8); -- implicit
        f(1) <= f(1) + xemu(1)(15 DOWNTO 8); -- divide by 2
    END IF;
END PROCESS Store;

MulGen1: FOR I IN 0 TO L-1 GENERATE
    p(i) <= f(i) * x(i);
END GENERATE;

y <= p(0) + p(1); -- Compute ADF output

ysxt: PROCESS (y) -- Scale y by 128 because x is fraction
BEGIN
    sxty(8 DOWNTO 0) <= y(15 DOWNTO 7);
    FOR k IN 15 DOWNTO 9 LOOP
        sxty(k) <= y(y'high);
    END LOOP;
END PROCESS;

e <= sxtd - sxty;
emu <= e(8 DOWNTO 1); -- e*mu divide by 2 and
-- 2 from xemu makes mu=1/4
MulGen2: FOR I IN 0 TO L-1 GENERATE
    xemu(i) <= emu * x(i);
END GENERATE;

y_out <= sxty; -- Monitor some test signals
e_out <= e;
f0_out <= f(0);
f1_out <= f(1);

END fpga;

```

The design is a literal interpretation of the adaptive LMS filter architecture found in Fig. 8.20 (p. 563). The output of each tap of the tapped delay line is multiplied by the appropriate filter coefficient and the results are added. The response of the adaptive filter y and of the overall system e to a reference signal x and a desired signal d is shown in Fig. 8.21. The filter adapts after approximately 20 steps at $1\mu s$ to the optimal values $f_0 = 43.3$ and $f_1 = 25$. The design uses 51LEs, four embedded multipliers, and has a registered performance of **Fmax**=69.26 MHz using the TimeQuest slow 85C model. 8.5

The previous example also shows that the standard LMS implementation has a low **Registered Performance** due to the fact that two multipliers and several add operations have to be performed in one clock cycle before the filter coefficient can be updated. In the following section we wish therefore to study how to achieve a higher throughput.

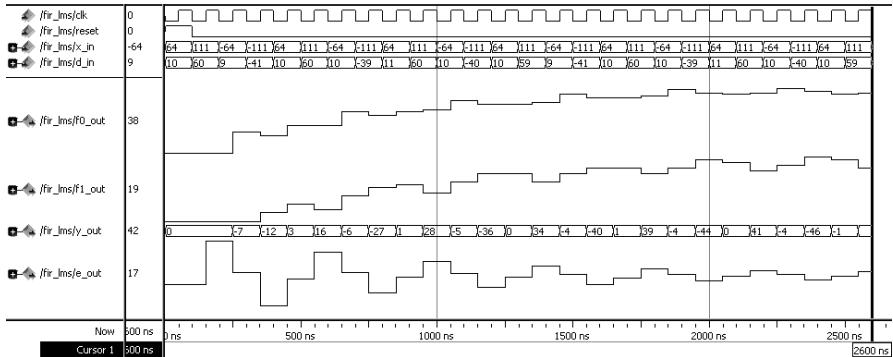


Fig. 8.21. VHDL simulation of the power-line interference cancellation using the LMS algorithm

8.5.3 Pipelined LMS Filters

As can be seen from Fig. 8.20 (p. 563) the original LMS adaptive filter has a long update path and hence the performance already for 8-bit data and coefficients is relatively slow. It is therefore no surprise that many attempts have been made to improve the throughput of the LMS adaptive filter. The optimal number of pipeline stages from Fig. 8.20 (p. 563) can be computed as follows: We use embedded multipliers that only need one output register for maximum performance, see also Fig. 2.15, p. 89. For the adder tree an additional $\log_2(L)$ pipeline stages would be sufficient and one additional stage for the computation of the error signal. The total number of pipeline stages for a maximum throughput are therefore

$$D_{\text{opt}} = \log_2(L) + 3, \quad (8.49)$$

where we have assumed that μ is a power-of-two constant and the scaling with μ can be done without the need of additional pipeline stages. If, however, the normalized LMS is used, then μ will no longer be a constant and depending on the bit width of μ additional pipeline stages will be required.

Pipelining an LMS filter is not as simple as for an FIR filter, because the LMS has, as the IIR filter, feedback. We need therefore to ensure that the coefficient of the pipelined filter still converges to the same coefficient as the adaptive filter without pipelining. Most of the ideas to pipeline IIR filters can be used to pipeline an LMS adaptive filter. The suggestion include

- Delayed LMS [279, 289, 290]
- Look-ahead transformation of the pipelined LMS [274, 291, 292]
- Transposed form LMS filter [293]
- Block transformation using FFTs [284]

We have already discussed the block transform algorithms and now wish in the following to briefly review the other techniques to improve the LMS throughput.

The delayed LMS algorithm. In the delayed LMS algorithm (DLMS) the assumption is that the gradient of the error $\nabla[n] = e[n]\mathbf{x}[n]$ does not change much if we delay the coefficient update by a couple of samples, i.e., $\nabla[n] \approx \nabla[n - D]$. It has been shown [289, 290] that as long as the delay is less than the system order, i.e., filter length, this assumption is well true and the update does not degrade the convergence speed. Long's original DLMS algorithm only considered pipelining the adder tree of the adaptive filter assuming also that multiplication and coefficient update can be done in one clock cycle (like for programmable digital signal processors [279]), but for a FPGA implementation multiplier and the coefficient update requires additional pipeline stages. If we introduce a delay of D_1 in the filter computation path and D_2 in the coefficient update path the LMS Algorithm 8.2 (p. 546) becomes:

$$\begin{aligned} e[n - D_1] &= d[n - D_1] - \mathbf{f}^T[n - D_1]\mathbf{x}[n - D_1] \\ \mathbf{f}[n + 1] &= \mathbf{f}[n - D_1 - D_2] + \mu e[n - D_1 - D_2]\mathbf{x}[n - D_1 - D_2]. \end{aligned}$$

The look-ahead DLMS algorithm. For long adaptive filters with $D = D_1 + D_2 < L$ the delayed coefficient update presented in the previous section, in general, does not change the convergence of the ADF much. It can, however, for shorter filters become necessary to reduce or even remove the change in system function completely. From the IIR pipelining method we have discussed in Chap. 4, the time domain interleaving method can always be applied. We perform just a look-ahead in coefficient computation, without alternating the overall system. Let us start with the DLMS update equations with pipelining only in the coefficient computation, i.e.,

$$\begin{aligned} e^{\text{DLMS}}[n - D] &= d[n - D] - \mathbf{x}^T[n - D]\mathbf{f}[n - D] \\ \mathbf{f}[n + 1] &= \mathbf{f}[n] + \mu e[n - D]\mathbf{x}[n - D]. \end{aligned}$$

But the error function of the LMS would be

$$e^{\text{LMS}}[n - D] = d[n - D] - \mathbf{x}^T[n]\mathbf{f}[n - D].$$

We follow the idea from Poltmann [291] and wish to compute the correction term $\Lambda[n]$, which cancels the change of the DLMS error computation compared with the LMS, i.e.,

$$\Lambda[n] = e^{\text{LMS}}[n - D] - e^{\text{DLMS}}[n - D].$$

The error function of the DLMS is now changed to

$$e^{\text{DLMS}}[n - D] = d[n - D] - \mathbf{x}^T[n - D]\mathbf{f}[n - D] - \Lambda[n].$$

We need therefore to determine the term

$$\Lambda[n] = \mathbf{x}^T[n - D](\mathbf{f}[n] - \mathbf{f}[n - D]).$$

The term in brackets can be recursively determined via

$$\begin{aligned}
 & \mathbf{f}[n] - \mathbf{f}[n - D] \\
 &= \mathbf{f}[n - 1] + \mu e[n - D - 1] \mathbf{x}[n - D_1] - \mathbf{f}[n - D] \\
 &= \mathbf{f}[n - 2] + \mu e[n - D - 2] \mathbf{x}[n - D - 2] \\
 &\quad + \mu e[n - D - 1] \mathbf{x}[n - D - 1] - \mathbf{f}[n - D] \\
 &= \sum_{s=1}^D \mu e[n - D - s] \mathbf{x}[n - D - s],
 \end{aligned}$$

and it follows for the correction term $\Lambda[n]$ finally

$$\begin{aligned}
 \Lambda[n] &= \mathbf{x}^T[n - D] \left(\sum_{s=1}^D \mu e[n - D - s] \mathbf{x}[n - D - s] \right) \\
 e^{\text{DLMS}}[n - D] &= d[n - D] - \mathbf{x}^T[n - D] \mathbf{f}[n - D] \\
 &\quad - \mathbf{x}^T[n - D] \left(\sum_{s=1}^D \mu e[n - D - s] \mathbf{x}[n - D - s] \right).
 \end{aligned}$$

It can be seen that this correction term needs an additional $2D$ multiplication, which may be too expensive in some applications. It has been suggested [292] to “relax” the requirement for the correction term but some additional multipliers are still necessary.

We can, however, remove the influence of the coefficient update delay, by applying the look-ahead principle [274], i.e.,

$$\mathbf{f}[n + 1] = \mathbf{f}[n - D_1] + \mu \sum_{k=0}^{D_2 - 1} e[n - D_1 - k] \mathbf{x}[n - D_1 - k]. \quad (8.50)$$

The summation in (8.50) builds the moving average over the last D_2 gradient values, and makes it intuitively clear that the convergence will proceed more smoothly. The advantage compared with the transformation from Poltmann is that this look-ahead computation can be done without a general multiplication. The moving average in (8.50) may even be implemented with a first order CIC filter (see Fig. 5.15, p. 320), which reduced the arithmetic effort to one adder and a subtractor.

Similar approaches to the idea from Poltmann to improve the DLMS algorithm have also been suggested [294–296].

8.5.4 Transposed Form LMS Filter

We have seen that the DLMS algorithm can be smoothed by introducing a look-ahead computation in the coefficient update, as we have used in IIR filters, but is, in general, not without additional cost. If we use, however, the transposed FIR structure (see Fig. 3.3, p. 181) instead of the direct structure, we can eliminate the delay by the adder tree completely. This will reduce the

requirement for the optimal number of pipeline stages from (8.49), p. 565, by $\log_2(L)$ stages. For a LTI system both direct and transposed filters are described by the same convolution equation, but for a time-varying coefficient we need to change the filter coefficient from

$$f_k[n] \quad \text{to} \quad f_k[n - k]. \quad (8.51)$$

The equation for the estimated gradient (8.17) on page 545 now becomes

$$\hat{\nabla}[n] = -2e[n] \frac{\partial e[n - k]}{\partial \mathbf{f}_k[n]} \quad (8.52)$$

$$= -2e[n] \mathbf{x}[n - k] \frac{f_k[n - k]}{\partial \mathbf{f}_k[n]}. \quad (8.53)$$

If we now assume that the coefficient update is relatively slow, i.e., $f_k[n - k] \approx f_k[n]$ the gradient becomes,

$$\hat{\nabla}[n] \approx -2e[n] \mathbf{x}[n], \quad (8.54)$$

and the coefficient update equation becomes:

$$f_k[n - k + 1] = f_k[n - k] + \mu e[n] \mathbf{x}[n]. \quad (8.55)$$

The learning characteristics of the transposed-form adaptive filter algorithms have been investigated by Jones [293], who showed that we will get a somewhat slower convergence rate when compared with the original LMS algorithm. The stability bound regarding μ also needs to be determined and is found to be smaller than for the LMS algorithm.

8.5.5 Design of DLMS Algorithms

If we wish to pipeline the LMS filter from Example 8.5 (p. 561) we conclude from the discussion above (8.49) that the optimal number of pipeline stages when using embedded multipliers becomes:

$$D_{\text{opt}} = \log_2(L) + 3 = 1 + 3 = 4. \quad (8.56)$$

On the other hand, pipelining LE-based $B \times B$ -bit multipliers would require additional $2 \times \log_2(B)$ pipeline stages; see [34]. Figure 8.22 shows a MATLAB simulation in 8-bit precision with a delay 4. Compared with the original LMS design from Example 8.5 (p. 561) it shows some “overswing” in the adaptation process.

Example 8.6: Two-Tap Pipelined Adaptive LMS FIR Filter

The VHDL design ⁴ for a filter with two coefficients f_0 and f_1 with a step size of $\mu = 1/4$ is shown in the following listing:

⁴ The equivalent Verilog code `fir4lms.v` for this example can be found in Appendix A on page 862. Synthesis results are shown in Appendix B on page 881.

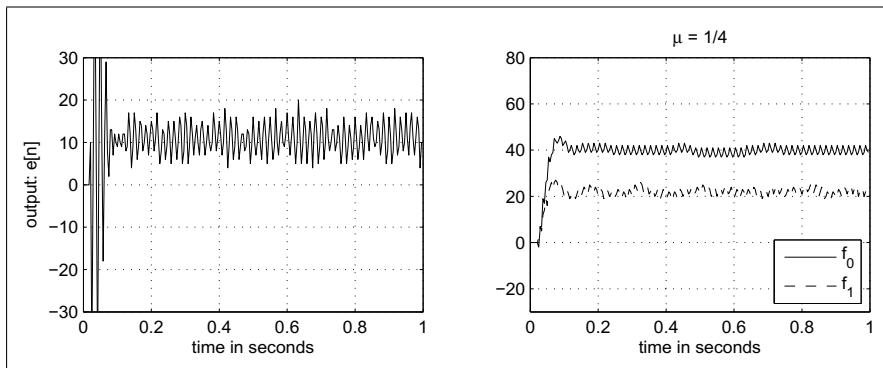


Fig. 8.22. 8-bit MATLAB simulation of the power-line interference cancellation using the DLMS algorithm with a delay of four

```
-- This is a generic DLMS FIR filter generator
-- It uses W1 bit data/coefficients bits

LIBRARY ieee; USE ieee.std_logic_1164.ALL;
PACKAGE n_bit_int IS                                -- User defined types
    CONSTANT W1 : INTEGER := 8;   -- Input bit width
    CONSTANT W2 : INTEGER := 16;  -- Multiplier bit width 2*W1
    SUBTYPE SLV1 IS STD_LOGIC_VECTOR(W1-1 DOWNTO 0);
    SUBTYPE SLV2 IS STD_LOGIC_VECTOR(W2-1 DOWNTO 0);
END n_bit_int;
LIBRARY work; USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-----
ENTITY fir4dlms IS                                -----> Interface
    GENERIC (L      : INTEGER := 2);   -- Filter taps
    PORT ( clk     : IN  STD_LOGIC;    -- System clock
           reset   : IN  STD_LOGIC;    -- Asynchronous reset
           x_in    : IN  SLV1;        -- System input
           d_in    : IN  SLV1;        -- Reference input
           f0_out  : OUT SLV1;       -- 1. filter coefficient
           f1_out  : OUT SLV1;       -- 2. filter coefficient
           y_out   : OUT SLV2;       -- System output
           e_out   : OUT SLV2);      -- Error signal
END fir4dlms;
-----
ARCHITECTURE fpga OF fir4dlms IS

    TYPE ARRAY_F IS ARRAY (0 TO L-1) OF SLV1;
    TYPE ARRAY_X IS ARRAY (0 TO 4) OF SLV1;
    TYPE ARRAY_D IS ARRAY (0 TO 2) OF SLV1;
    TYPE AO_L1SLV2 IS ARRAY (0 TO L-1) OF SLV2;
```

```

SIGNAL xemu0, xemu1 : SLV1;
SIGNAL emu      : SLV1;
SIGNAL y, sxty : SLV2;

SIGNAL e, sxtd : SLV2;
SIGNAL f        : ARRAY_F; -- Coefficient array
SIGNAL x        : ARRAY_X; -- Data array
SIGNAL d        : ARRAY_D; -- Reference array
SIGNAL p, xemu : A0_L1SLV2; -- Product array

BEGIN

dsxt: PROCESS (d) -- make d a 16 bit number
BEGIN
    sxtd(7 DOWNTO 0) <= d(2);
    FOR k IN 15 DOWNTO 8 LOOP
        sxtd(k) <= d(2)(7);
    END LOOP;
END PROCESS;

Store: PROCESS (clk, reset) -----> Store these data or
BEGIN                                -- coefficients in registers
    IF reset = '1' THEN              -- Asynchronous clear
        FOR k IN 0 TO 2 LOOP
            d(k) <= (OTHERS => '0');
        END LOOP;
        FOR k IN 0 TO 4 LOOP
            x(k) <= (OTHERS => '0');
        END LOOP;
        FOR k IN 0 TO 1 LOOP
            f(k) <= (OTHERS => '0');
        END LOOP;
    ELSIF rising_edge(clk) THEN
        d(0) <= d_in;   -- Shift register for desired data
        d(1) <= d(0);
        d(2) <= d(1);
        x(0) <= x_in;   -- Shift register for data
        x(1) <= x(0);
        x(2) <= x(1);
        x(3) <= x(2);
        x(4) <= x(3);
        f(0) <= f(0) + xemu(0)(15 DOWNTO 8); -- implicit
        f(1) <= f(1) + xemu(1)(15 DOWNTO 8); -- divide by 2
    END IF;
END PROCESS Store;

Mul: PROCESS (clk, reset) -----> Store these data or
BEGIN                                -- coefficients in registers
    IF reset = '1' THEN              -- Asynchronous clear
        FOR k IN 0 TO L-1 LOOP
            p(k) <= (OTHERS => '0');
            xemu(k) <= (OTHERS => '0');
        END LOOP;
    END IF;
END PROCESS Mul;

```

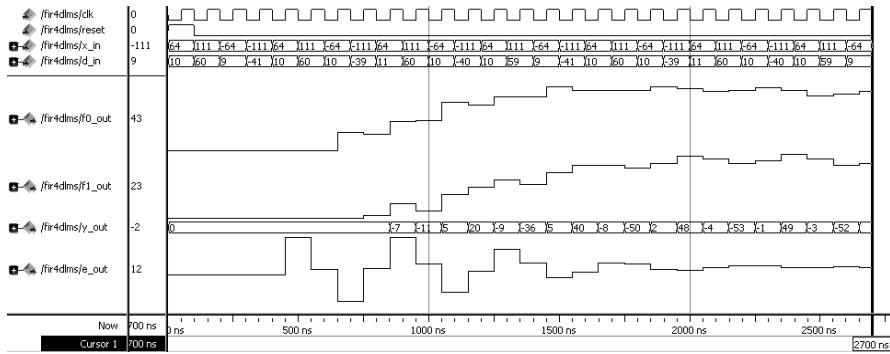


Fig. 8.23. VHDL simulation of the power-line interference cancelation using the DLMS algorithm with a delay of four

```

        END LOOP;
y <= (OTHERS => '0');
e <= (OTHERS => '0');
ELSIF rising_edge(clk) THEN
    FOR I IN 0 TO L-1 LOOP
        p(i) <= f(i) * x(i);
        xemu(i) <= emu * x(i+3);
    END LOOP;
    y <= p(0) + p(1); -- Computer ADF output:log(L) adds
    e <= sxtd - sxtys; -- e*mu divide by 2 and 2
    END IF;
END PROCESS Mul;

emu <= e(8 DOWNTO 1); -- from xemu makes mu=1/4

ysxt: PROCESS (y) -- scale y by 128 because x is fraction
BEGIN
    sxtys(8 DOWNTO 0) <= y(15 DOWNTO 7);
    FOR k IN 15 DOWNTO 9 LOOP
        sxtys(k) <= y(y'high);
    END LOOP;
END PROCESS;

y_out <= sxtys; -- Monitor some test signals
e_out <= e;
f0_out <= f(0);
f1_out <= f(1);

END fpga;

```

The design is a literal interpretation of the adaptive LMS filter architecture found in Fig. 8.20 (p. 547) with one additional delay of pipeline stages for each multiplier and the two adders. The output of each tap of the tapped delay line is multiplied by the appropriate filter coefficient and the results are added. Note the additional delays for x and d in the **Store: PROCESS** to

make the signals coherent. The response of the adaptive filter y and of the overall system e to a reference signal x and a desired signal d is shown in the VHDL simulation in Fig. 8.23. The filter adapts after approximately 20 steps at $2\ \mu s$ to the optimal values $f_0 = 43.3$ and $f_1 = 25$, but f_0 also shows some overshwing in the adaptation process. The design uses 106 LEs, four embedded multipliers, and has a registered performance of $F_{max}=261.57$ MHz using the TimeQuest slow 85C model.

8.6

If we compare the pipeline LMS with the original LMS algorithm we may gain up to a factor of 4 speed improvement, while at the same time twice the LEs are needed. The additional effort comes from the extra delays of the reference data $d[n]$ and the filter input $x[n]$. The limitation is just that it may become necessary for large pipeline delays to adjust μ in order to guarantee stability.

8.5.6 LMS Designs using SIGNUM Function

We saw in the previous section that the implementation cost of the LMS algorithm is already high for short filter length. The highest cost of the filter comes from the large number of general multipliers and the major goal in reducing the effort is to reduce the number of multipliers. Obviously the FIR filter part cannot be reduced, but different simplifications in the computation of the coefficient update have been investigated. Given the fact that to ensure stability usually the step size is chosen much smaller than μ_{max} , the following suggestions have been made:

- Use only the sign of the reference *data* $x[n]$ not the full precision value to update the filter coefficients.
- Use only the sign of the *error* $e[n]$ not the full precision value to update the filter coefficients.
- Use both of the previous simplifications via the sign of error and data.

The three modifications can be described with the following coefficient update equations in the LMS algorithm:

$$\begin{aligned} \mathbf{f}[n+1] &= \mathbf{f}[n] + \mu \times e[n] \times \text{sign}(\mathbf{x}[n]) && \text{sign data function} \\ \mathbf{f}[n+1] &= \mathbf{f}[n] + \mu \times \mathbf{x}[n] \times \text{sign}(e[n]) && \text{sign error function} \\ \mathbf{f}[n+1] &= \mathbf{f}[n] + \mu \times \text{sign}(e[n]) \times \text{sign}(\mathbf{x}[n]) && \text{sign-sign function}. \end{aligned}$$

We note from the simulation of the three possible simplifications shown in Fig. 8.24 that for the sign-data function almost the same result occurs as in the full precision case. This is no surprise, because our input reference signal $x[n] = \cos[\pi n/2 + \phi]$ will not be much quantized through the sign operation anyway. This is much different for the sign-error function. Here the quantization through the sign operation essentially alters the time

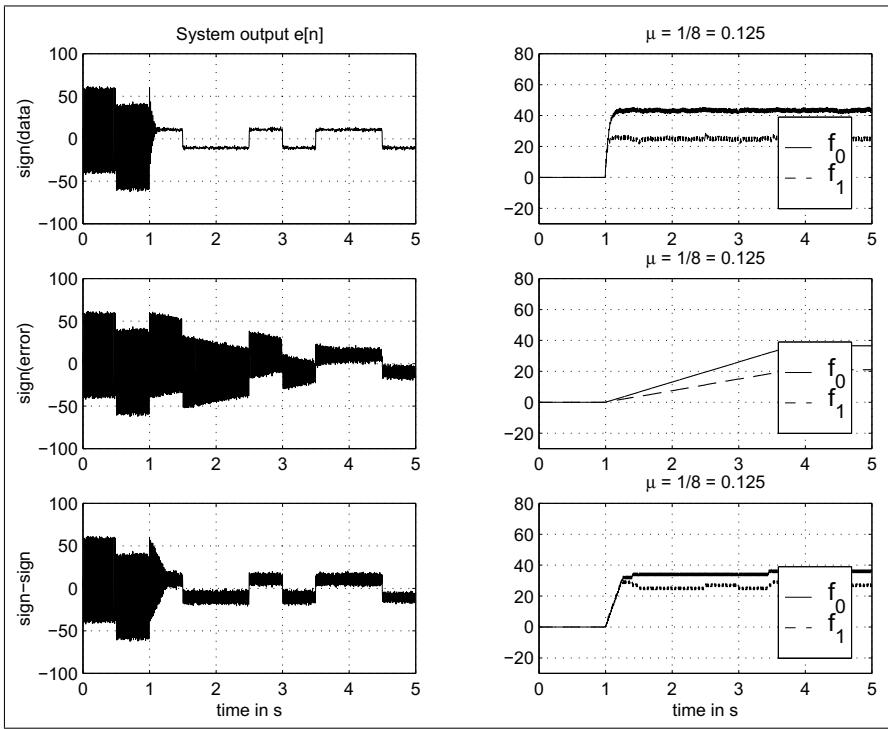


Fig. 8.24. Simulation of the power-line interference cancellation using the 3 simplified signed LMS (SLMS) algorithms. System output $e[n]$. (left); filter coefficients (right)

constant of the system. But finally, after about 2.5 s the correct values are reached, although from the system output $e[n]$ we note the essential ripple in the output function even after a long simulation time. Finally, the sign-sign algorithm converges faster than the sign-error algorithm, but here also the system output shows essential ripple for $e[n]$. From the simulation it can be seen that the sign-function simplification (to save the L multiplications in the filter coefficient update) has to be evaluated carefully for the specific application to still guarantee a stable system and acceptable time constants of the system. In fact, it has been shown that for specific signals and application the sign algorithms does not converge, although the full precision algorithm would converge. Besides the sign effect we also need to ensure that the integer quantization through the implementation does not alter the desired system properties.

Another point to consider when using the sign function is the error floor that can be reached. This is discussed in the following example.

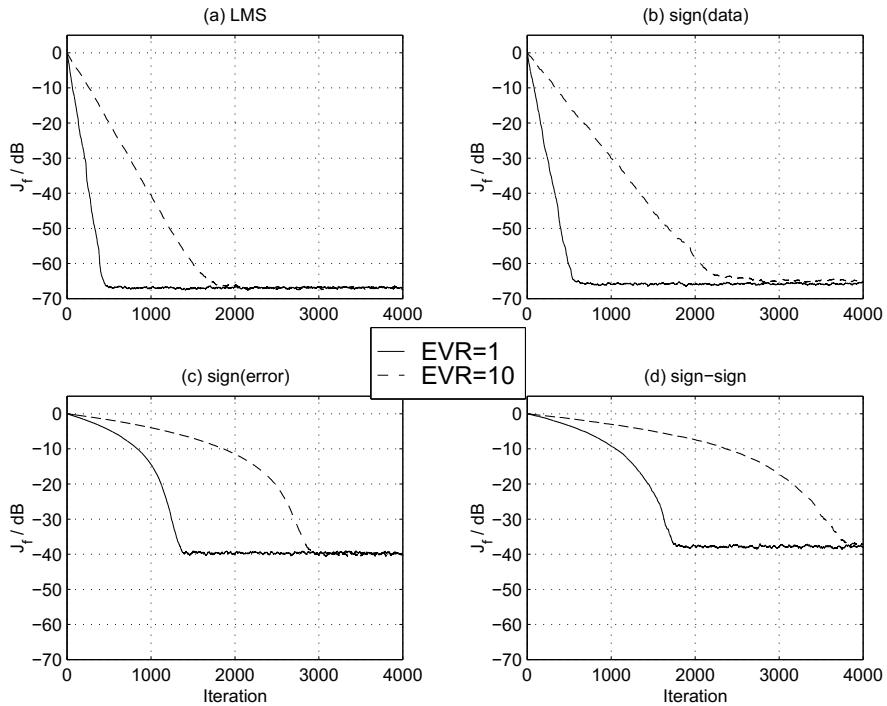


Fig. 8.25. Simulation of the system identification experiment using the 3 simplified signed LMS algorithms for an average of 50 learning curves for an error floor of -60 dB. (a) LMS with full precision. (b) signed data. (c) signed error algorithms. (d) sign-sign LMS algorithm

Example 8.7: Error Floor in Signum LMS Filters

Suppose we have a system identification configuration as discussed in Sect. 8.3.1 (p. 551), and we wish to use one of the signum-type ADF algorithms. What will then be the error floor that can be reached? Obviously through the signum operation we will lose some precision and we expect that we will not reach the same low-noise level as with a full-precision LMS algorithm. We also expect that the learning rate will be somewhat decreased when compared with the full-precision LMS algorithm. This can be verified by the simulation results shown in Fig. 8.25 for an average over 50 learning curves and two different eigenvalue ratios (EVRs). The sign data algorithms shows some delay in the adaptation when compared with the full-precision LMS algorithm, but reaches the error floor, which was set to -60 dB. Signed error and sign-sign algorithms show larger delays in the adaptation and also reach only an error floor of about -40 dB. This larger error may or may not be acceptable for some applications.

8.7

The sign-sign algorithm is attractive from a software or hardware implementation standpoint and has been used for the International Telecommuni-

cation Union (ITU) standard for adaptive differential pulse code modulation (ADPCM) transmission. From a hardware implementation standpoint we actually do not need to implement the sign-sign algorithm, because the multiplication with μ is just a scaling with a constant and one of the single sign algorithms will already allow us to save the L multipliers we usually need for the filter coefficient update in Fig. 8.20 (p. 563).

8.6 Recursive Least Square Algorithms

In the LMS algorithm we have discussed in the previous sections the filter coefficients are gradually adjusted by a stochastic gradient method to finally approximate the Wiener–Hopf optimal solution. The recursive least square (RLS) algorithm takes another approach. Here, the estimation of the $(L \times L)$ autocorrelation matrix $\mathbf{R}_{\mathbf{x}\mathbf{x}}$ and the cross-correlation vector $\mathbf{r}_{d\mathbf{x}}$ are iteratively updated with each new incoming data pair $(x[n], d[n])$. The simplest approach would be to reconstruct the Wiener–Hopf equation (8.12), i.e., $\mathbf{R}_{\mathbf{x}\mathbf{x}}\mathbf{f}_{\text{opt}} = \mathbf{r}_{d\mathbf{x}}$ and resolve it. However, this would be the equivalent of one matrix inversion as each new data point pair arrives and has the potential of being computationally expensive. The main goal of the RLS algorithms we will discuss in the following is therefore to seek a (iterative) time recursion for the filter coefficients $\mathbf{f}[n+1]$ in terms of the previous least square estimate $\mathbf{f}[n]$ and the new data pair $(x[n], d[n])$. Each incoming new value $x[n]$ is placed in the length- L data array $\mathbf{x}[n] = [x[n] x[n-1] \dots x[n-(L-1)]]^T$. We then wish to add $x[n]x[n]$ to $\mathbf{R}_{\mathbf{x}\mathbf{x}}[0, 0]$, $x[n]x[n-1]$ to $\mathbf{R}_{\mathbf{x}\mathbf{x}}[0, 1]$, etc. Mathematically we just compute the outer product $\mathbf{x}\mathbf{x}^T$ and add this $(L \times L)$ matrix to the previous estimation of the autocorrelation matrix $\mathbf{R}_{\mathbf{x}\mathbf{x}}[n]$. The recursive computation may be computed as follows:

$$\mathbf{R}_{\mathbf{x}\mathbf{x}}[n+1] = \mathbf{R}_{\mathbf{x}\mathbf{x}}[n] + \mathbf{x}[n]\mathbf{x}^T[n] = \sum_{s=0}^n \mathbf{x}[s]\mathbf{x}^T[s]. \quad (8.57)$$

For the cross-correlation vector $\mathbf{r}_{d\mathbf{x}}[n+1]$ we also build an “improved” estimate by adding with each new pair $(x[n], d[n])$ the vector $d[n]\mathbf{x}[n]$ to the previous estimation of $\mathbf{r}_{d\mathbf{x}}[n]$. The recursion for the cross-correlation becomes

$$\mathbf{r}_{d\mathbf{x}}[n+1] = \mathbf{r}_{d\mathbf{x}}[n] + d[n]\mathbf{x}[n], \quad (8.58)$$

we can now use the Wiener–Hopf equation in a time recursive fashion and compute

$$\mathbf{R}_{\mathbf{x}\mathbf{x}}[n+1]\mathbf{f}_{\text{opt}}[n+1] = \mathbf{r}_{d\mathbf{x}}[n+1]. \quad (8.59)$$

For the true estimates of cross- and autocorrelation matrices we would need to scale by the number of summations, which is proportional to n , but the cross- and autocorrelation matrices are scaled by the same factor, which cancel each other out in the iterative algorithm and we get for the filter coefficient update

$$\mathbf{f}_{\text{opt}}[n+1] = \mathbf{R}_{xx}^{-1}[n+1]\mathbf{r}_{dx}[n+1]. \quad (8.60)$$

Although this first version of the RLS algorithms is computationally intensive (approximately L^3 operations are needed for the matrix inversion) it still shows the principal idea of the RLS algorithm and can be quickly programmed, for instance in MATLAB, as the following code segment shows the inner loop for length- L RLS filter algorithm:

```

x = [xin;x(1:L-1)]; % get new sample
y = f' * x;           % filter output
err = din - y;        % error: reference - filter output
Rxx = Rxx + x*x';    % update the autocorrelation matrix
rdx = rdx + din .* x; % update the cross-correlation vector
f = Rxx^(-1) * rdx;  % compute filter coefficients

```

where \mathbf{R}_{xx} is a $(L \times L)$ matrix and \mathbf{rdx} is a $(L \times 1)$ vector. The cross-correlation vector is usually initialized with $\mathbf{r}_{dx}[0] = \mathbf{0}$. The only problem with the algorithm so far arises at the first $n < L$ iterations, when $\mathbf{R}_{xx}[n]$ only has a few nonzero entries, and consequently will be singular and no inverse exists. There are a couple of ways to tackle this problem:

- We can wait with the computation of the inverse until we find that the autocorrelation matrix is nonsingular, i.e., $\det(\mathbf{R}_{xx}[n]) > 0$.
- We can use $\mathbf{R}_{xx}^+[n] = (\mathbf{R}_{xx}^T[n]\mathbf{R}_{xx}[n])^{-1}\mathbf{R}_{xx}^T[n]$ the so-called pseudoinverse, which is a standard result in linear algebra regarding the solution of an overdetermined set of linear equations.
- We can initialize the autocorrelation matrix \mathbf{R}_{xx} with $\delta\mathbf{I}$ where δ is chosen to be a small (large) constant for high (low) S/N ratio of the input signal.

The third approach is the most popular due to the computational benefit and the possibility to set an initial “learning rate” using the constant δ . The influence of the initialization in the RLS algorithm for an experiment similar to Sect. 8.3.1 (p. 551) with an average over 5 learning curves is shown in Fig. 8.26. The upper row shows the full-length simulation over 4000 iterations, while the lower row shows the first 100 iterations only. For high S/N (-48 dB) we may use a large value for the initialization, which yields a fast convergence. For low S/N values (-10 dB) small initialization values should be used, otherwise large errors at the first iterations can occur, which may or may not be tolerable for the specific application.

A more computationally attractive approach than the first “brute force” RLS algorithm will be discussed in the following. The key idea is that we do not compute the matrix inversion at all and use a time recursion directly for $\mathbf{R}_{xx}^{-1}[n]$, we actually will never have (or need) $\mathbf{R}_{xx}[n]$ available. To do so, we substitute the Wiener equation for time $n+1$, i.e., $\mathbf{f}[n+1]\mathbf{R}_{xx}[n+1] = \mathbf{r}_{dx}[n+1]$ into (8.58) it follows that:

$$\mathbf{R}_{xx}[n+1]\mathbf{f}[n+1] = \mathbf{R}_{xx}[n]\mathbf{f}[n] + d[n+1]\mathbf{x}[n+1]. \quad (8.61)$$

Now we use (8.57) to get

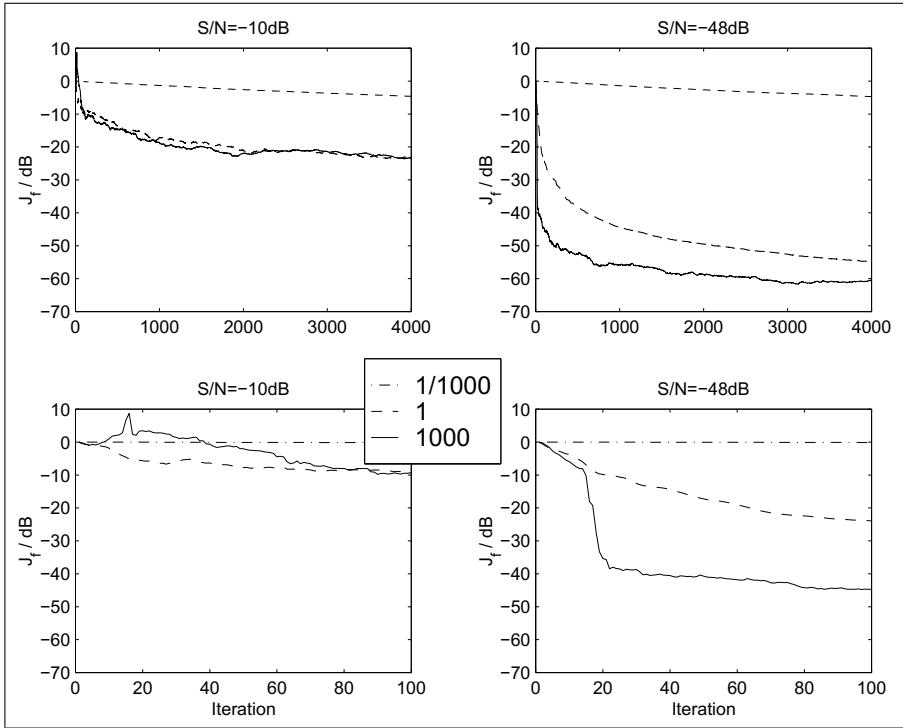


Fig. 8.26. Learning curves of the RLS algorithms using different initialization of $\mathbf{R}_{\mathbf{xx}}^{-1}[0] = \delta \mathbf{I}$ or $\mathbf{R}_{\mathbf{xx}}[0] = \delta^{-1} \mathbf{I}$. High S/N is -48 dB and low is -10 dB. $\delta = 1000, 1$ or $1/1000$

$$\begin{aligned} \mathbf{R}_{\mathbf{xx}}[n+1]\mathbf{f}[n+1] &= (\mathbf{R}_{\mathbf{xx}}[n+1] - \mathbf{x}[n+1]\mathbf{x}^T[n+1])\mathbf{f}[n] \\ &\quad + d[n+1]\mathbf{x}[n+1]. \end{aligned} \quad (8.62)$$

We can rearrange (8.62) by multiplying by $\mathbf{R}_{\mathbf{xx}}^{-1}[n+1]$ to have $\mathbf{f}[n+1]$ on the lefthand side of the equation:

$$\begin{aligned} \mathbf{f}[n+1] &= \mathbf{f}[n] + \underbrace{\mathbf{R}_{\mathbf{xx}}^{-1}[n+1]\mathbf{x}[n+1]}_{\mathbf{k}[n+1]} \underbrace{(d[n+1] - \mathbf{f}^T[n]\mathbf{x}[n+1])}_{e[n+1]} \\ &= \mathbf{f}[n] + \mathbf{k}[n+1]e[n+1], \end{aligned}$$

where the *a priori error* is defined as

$$e[n+1] = d[n+1] - \mathbf{f}^T[n]\mathbf{x}[n+1],$$

and the *Kalman gain vector* is defined as

$$\mathbf{k}[n+1] = \mathbf{R}_{\mathbf{xx}}^{-1}[n+1]\mathbf{x}[n+1]. \quad (8.63)$$

As mentioned above the direct computation of the matrix inversion is computationally intensive, and it is much more efficient to use again the iteration

equation (8.57) to actually avoid the inversion at all. We use the so-called “matrix inversion lemma,” which can be written as the following matrix identity:

$$\begin{aligned} & (\mathbf{A} + \mathbf{BCD})^{-1} \\ &= \mathbf{A}^{-1} - (\mathbf{A}^{-1}\mathbf{B}\mathbf{D}\mathbf{A}^{-1})(\mathbf{C}^{-1} + \mathbf{D}\mathbf{A}^{-1}\mathbf{B})^{-1}, \end{aligned}$$

which holds for all matrices \mathbf{A} , \mathbf{B} , \mathbf{C} , and \mathbf{D} , of compatible dimensions and nonsingular \mathbf{A} . We make the following associations:

$$\begin{array}{ll} \mathbf{A} = \mathbf{R}_{\mathbf{xx}}[n] & \mathbf{B} = \mathbf{x}[n] \\ \mathbf{C} = 1 & \mathbf{D} = \mathbf{x}^T[n]. \end{array}$$

The iterative equation for $\mathbf{R}_{\mathbf{xx}}^{-1}$ becomes:

$$\begin{aligned} \mathbf{R}_{\mathbf{xx}}^{-1}[n+1] &= (\mathbf{R}_{\mathbf{xx}}[n] + \mathbf{x}[n]\mathbf{x}^T[n])^{-1} \\ &= \mathbf{R}_{\mathbf{xx}}^{-1}[n] - \frac{\mathbf{R}_{\mathbf{xx}}^{-1}[n]\mathbf{x}[n]\mathbf{x}^T[n]\mathbf{R}_{\mathbf{xx}}^{-1}[n]}{1 + \mathbf{x}^T[n]\mathbf{R}_{\mathbf{xx}}^{-1}[n]\mathbf{x}[n]}. \end{aligned} \quad (8.64)$$

If we use the Kalman gain factor $\mathbf{k}[n]$ from (8.63) we can rewrite (8.64) more compactly as:

$$\mathbf{R}_{\mathbf{xx}}^{-1}[n+1] = \mathbf{R}_{\mathbf{xx}}^{-1}[n] - \frac{\mathbf{k}[n]\mathbf{k}^T[n]}{1 + \mathbf{x}^T[n]\mathbf{k}[n]}.$$

This recursion is as mentioned before initialized [268] with

$$\mathbf{R}_{\mathbf{xx}}^{-1}[0] = \delta \mathbf{I} \quad \text{with} \quad \delta = \begin{cases} \text{large positive constant for high SNR} \\ \text{small positive constant for low SNR.} \end{cases}$$

With this recursive computation of the inverse autocorrelation matrix the computation effort is now proportional to L^2 , an essential saving for large values of L . Figure 8.27 shows a summary of the RLS adaptive filter algorithm.

8.6.1 RLS with Finite Memory

As we can see from (8.57) and (8.58) the adaptive algorithm derived so far has an infinite memory. The values of the filter coefficients are functions of all past inputs starting with time zero. As will be discussed next it is often useful to introduce a “forgetting factor” into the algorithm, so that recent data are given greater importance than older data. This not only reduces the influence of older data, it also accomplishes that through the update of the cross- and autocorrelation with each new incoming data pair no overflow in the arithmetic will occur. One way of accomplishing a finite memory is to replace the sum-of-squares cost function, by an exponentially weighted sum of the output:

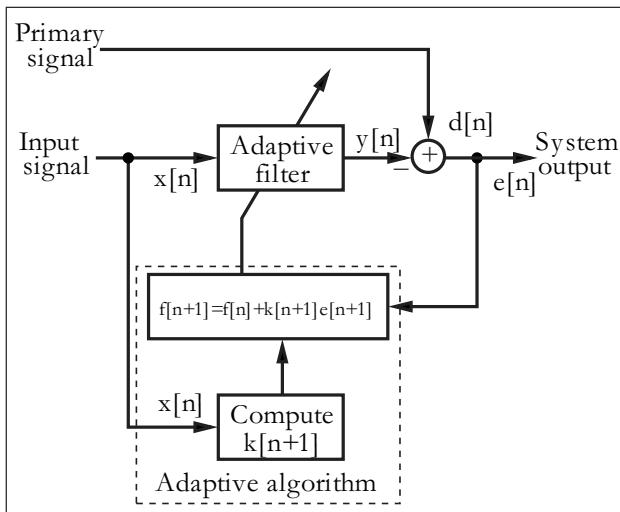


Fig. 8.27. Basic configuration for interference cancellation using the RLS algorithm

$$J = \sum_{s=0}^n \rho^{n-s} e^2[s], \quad (8.65)$$

where $0 \leq \rho \leq 1$ is a constant determining the effective memory of the algorithm. The case $\rho = 1$, is the infinite-memory case, as before. When $\rho < 1$ the algorithm will have an effective memory of $\tau = -1/\log(\rho) \approx 1/(1-\rho)$ data points. The exponentially weighted RLS algorithm can now be summarized as:

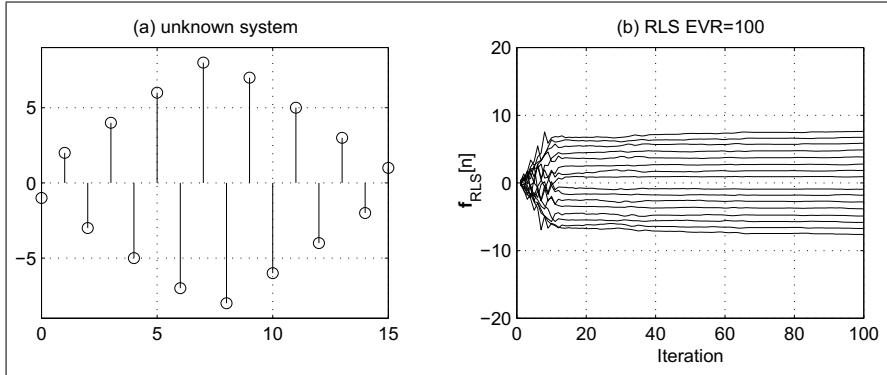


Fig. 8.28. Simulation of the $L = 16$ tap adaptive filter system identification. (a) Impulse response of the “unknown system.” (b) RLS coefficient learning curves for EVR = 100

Algorithm 8.8: RLS Algorithm

The exponentially weighted RLS algorithm to adjust the L coefficients of an adaptive filter uses the following steps:

- 1) Initialize $\mathbf{x} = \mathbf{f} = [0, 0, \dots, 0]^T$ and $\mathbf{R}_{\mathbf{xx}}^{-1}[0] = \delta \mathbf{I}$.
- 2) Accept a new pair of input samples $\{x[n+1], d[n+1]\}$ and shift $x[n+1]$ input the reference signal vector $\mathbf{x}[n+1]$.
- 3) Compute the output signal of the FIR filter, via
$$y[n+1] = \mathbf{f}^T[n] \mathbf{x}[n+1]. \quad (8.66)$$
- 4) Compute the a priori error function with
$$e[n+1] = d[n+1] - y[n+1]. \quad (8.67)$$
- 5) Compute the Kalman gain factor with
$$\mathbf{k}[n+1] = \mathbf{R}_{\mathbf{xx}}^{-1}[n+1] \mathbf{x}[n+1]. \quad (8.68)$$
- 6) Update the filter coefficient according to
$$\mathbf{f}[n+1] = \mathbf{f}[n] + \mathbf{k}[n+1] e[n+1]. \quad (8.69)$$
- 7) Update the filter inverse autocorrelation matrix according to
$$\mathbf{R}_{\mathbf{xx}}^{-1}[n+1] = \frac{1}{\rho} \left(\mathbf{R}_{\mathbf{xx}}^{-1}[n] - \frac{\mathbf{k}[n+1] \mathbf{k}^T[n+1]}{\rho + \mathbf{x}^T[n+1] \mathbf{k}[n+1]} \right). \quad (8.70)$$

Next continue with step 2.

The computational cost of the RLS are $(3L^2 + 9L)/2$ multiplications and $(3L^2 + 5L)/2$ additions or subtractions, per input sample, which is still more essential than the LMS algorithm. The advantage as we will see in the following example will be a higher rate of convergence and no need to select the step size μ , which may at times be difficult when stability of the adaptive algorithm has to be guaranteed.

Example 8.9: RLS Learning Curves

In this example we wish to evaluate a configuration called system identification to compare RLS and LMS convergence. We have used this type of

performance evaluation already for LMS ADF in Sect. 8.3.1 (p. 551) The system configuration is shown in Fig. 8.12 (p. 551). The adaptive filter has a length of $L = 16$, the same length as the “unknown” system, whose coefficients have to be learned. The additive noise level behind the “unknown system” has been set to -48 dB equivalent for an 8-bit quantization. For the LMS algorithm the eigenvalue ratio (EVR) is the critical parameter that determines the convergence speed, see (8.28), p. 548. In order to generate a different eigenvalue ratio we use a white Gaussian noise source with $\sigma^2 = 1$ that is filtered by a FIR type filter shown in Table 8.2 (p. 552). The coefficients are normalized to $\sum_k h[k]^2 = 1$, so that the signal power does not change. The impulse response of the unknown system is an odd filter with coefficients $1, -2, 3, -4, \dots, -3, 2, -1$ as shown in Fig. 8.28a. The step size for the LMS algorithm has been determined with

$$\mu_{\max} = \frac{2}{3 \times L \times E\{\mathbf{x}^2\}} = \frac{1}{24}. \quad (8.71)$$

In order to guarantee perfect stability the step size for the LMS algorithm has been chosen to be $\mu = \mu_{\max}/2 = 1/48$. For the transform-domain DCT-LMS algorithm a power normalization for each coefficient is used; see Fig. 8.17 (p. 559). From the simulation results shown in Fig. 8.29 it can be seen that the RLS converges faster than the LMS with increased EVR. DCT-LMS converges faster than LMS and in some cases quite as fast as the RLS algorithm. The DCT-LMS algorithm has less good performance when we look at the residue-error level and consistency of convergence. For higher EVR the RLS performance is better for both level and consistency of convergence. For EVR=1 the DCT-LMS reaches the value in the 50 dB range, but for EVR = 100 only 40 dB are reached. The RLS converges below the system noise. 8.9

8.6.2 Fast RLS Kalman Implementation

For the least-square FIR fast Kalman algorithm first presented by Ljung et al. [297] the concept of single-step linear forward and backward prediction play a central role. Using these forward and backward coefficients in an all-recursive, one-dimensional Levison–Durbin type algorithm it will be possible to update the Kalman gain vector with only an $O(L)$ type effort.

A one-step *forward* predictor is presented in Fig. 8.30. The predictor estimates the present value $x[n]$ based on its L most recent past values. The *a posteriori* error in the prediction is quantified by

$$\epsilon_L^f[n] = x[n] - \hat{x}[n] = x[n] - \mathbf{a}^T[n] \mathbf{x}_L[n-1]. \quad (8.72)$$

The superscript indicates that it is the forward prediction error, while the subscript describes the order (i.e., length) of the predictor. We will drop the index L and the vector length should be L for the remainder of this section, if not otherwise noted. It is also advantageous to compute also the *a priori* error that is computed using the filter coefficient of the previous iteration, i.e.,

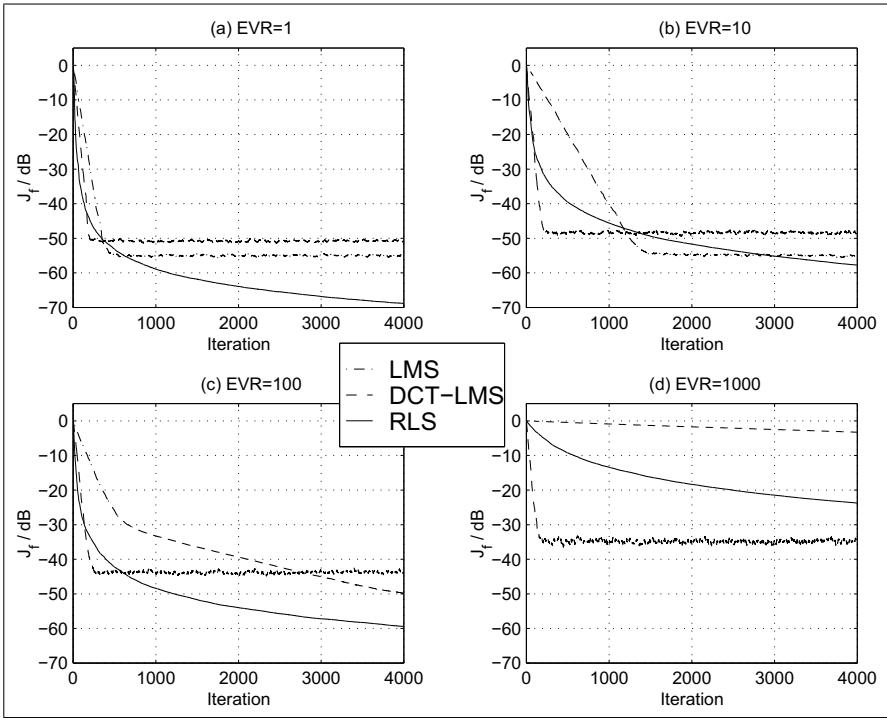


Fig. 8.29. Simulation results for a $L = 16$ -tap adaptive filter system identification. Learning curve J for LMS, transform-domain DCT-LMS, and RLS with $\mathbf{R}_{\mathbf{x}\mathbf{x}}^{-1}[0] = \mathbf{I}$. (a) EVR = 1. (b) EVR = 10. (c) EVR = 100. (d) EVR = 1000

$$e_L^f[n] = x[n] - \mathbf{a}^T[n-1]\mathbf{x}_L[n-1]. \quad (8.73)$$

The least-square minimum of $e_L^f[n]$ can be computed via

$$\frac{\partial(e_L^f[n])^2}{\partial \mathbf{a}^T[n]} = -E\{(x[s] - \mathbf{a}^T[s]\mathbf{x}[n])\mathbf{x}[n-s]\} = 0 \quad (8.74)$$

for $s = 1, 2, \dots, L$.

This leads again to an equation with the $(L \times L)$ autocorrelation matrix, but the right-hand side is different from the Wiener–Hopf equation:

$$\mathbf{R}_{\mathbf{x}\mathbf{x}}[n-1]\mathbf{a}[n] = \mathbf{r}^f[n] = \sum_{s=0}^n \mathbf{x}[s-1]\mathbf{x}[s]. \quad (8.75)$$

The minimum value of the cost function is given by

$$\alpha^f[n] = r_0^f[n] - \mathbf{a}^T[n]\mathbf{r}^f[n], \quad (8.76)$$

where $r_0^f[n] = \sum_{s=0}^n x[s]^2$.

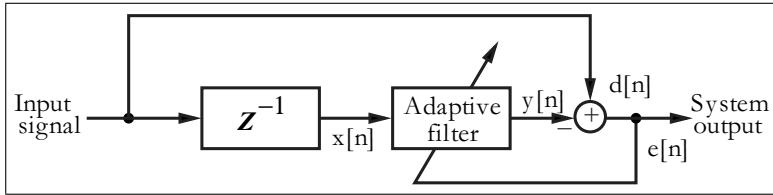


Fig. 8.30. Linear forward prediction of order L

The important fact about this predictor is now that the Levinson–Durbin algorithm can solve the least-square error minimum of (8.72) in a recursive fashion, without computing a matrix inverse. To update the predictor coefficient we need the same Kalman gain factor as in (8.69) for updating the filter coefficients, namely

$$\mathbf{a}_L[n+1] = \mathbf{a}_L[n] + \mathbf{k}_L[n]e_L^f[n].$$

We will see later how the linear prediction coefficients can be used to iteratively update the Kalman gain factor. In order to take advantage of the fact that the data vectors from one iteration to the next only differ in the first and last element, we use an augmented-by-one version $\mathbf{k}_{L+1}[n]$ of the Kalman gain update equation (8.68) which is given by

$$\mathbf{k}_{L+1}[n+1] = \mathbf{R}_{\mathbf{xx},L+1}^{-1}[n+1]\mathbf{x}_{L+1}[n+1]. \quad (8.77)$$

$$= \begin{bmatrix} \mathbf{r}_{0L}^f[n+1] & | & \mathbf{r}_L^{fT}[n+1] \\ \mathbf{r}_L^f[n] & | & \mathbf{R}_{\mathbf{xx},L}^{-1}[n] \end{bmatrix} \begin{bmatrix} \mathbf{x}[n+1] \\ \mathbf{x}_L[n] \end{bmatrix}. \quad (8.78)$$

In order to compute the matrix inverse of $\mathbf{R}_{\mathbf{xx},L+1}^{-1}[n]$ we use a well-known theorem of matrix inversion of block matrices, i.e.,

$$\mathbf{M}^{-1} = \begin{bmatrix} \mathbf{A} & | & \mathbf{B} \\ \mathbf{C} & | & \mathbf{D} \end{bmatrix}^{-1} = \quad (8.79)$$

$$\left[\frac{-(\mathbf{AD}^{-1}\mathbf{C} - \mathbf{A})^{-1}}{\mathbf{D}^{-1}\mathbf{C} - (\mathbf{AD}^{-1}\mathbf{C} - \mathbf{A})^{-1}} \middle| \frac{(\mathbf{AD}^{-1}\mathbf{C} - \mathbf{A})^{-1}\mathbf{BD}^{-1}}{\mathbf{D}^{-1} - (\mathbf{D}^{-1}\mathbf{CBD}^{-1})(\mathbf{AD}^{-1}\mathbf{C} - \mathbf{A})^{-1}} \right],$$

if \mathbf{D}^{-1} is nonsingular. We now make the following associations:

$$\mathbf{A} = \mathbf{r}_{0L}^f[n+1] \quad \mathbf{B} = \mathbf{r}_L^{fT}[n+1]$$

$$\mathbf{C} = \mathbf{r}_L^f[n] \quad \mathbf{D} = \mathbf{R}_{\mathbf{xx}}^{-1}[n],$$

we then get

$$\begin{aligned} \mathbf{D}^{-1}\mathbf{C} &= \mathbf{R}_{\mathbf{xx},L}^{-1}[n]\mathbf{r}_L^f[n] = \mathbf{a}_L[n+1] \\ \mathbf{BD}^{-1} &= \mathbf{r}_L^{fT}[n+1]\mathbf{R}_{\mathbf{xx}}^{-1}[n] = \mathbf{a}_L^T[n+1] \\ -(\mathbf{AD}^{-1}\mathbf{C} - \mathbf{A})^{-1} &= -\mathbf{r}_L^{fT}[n+1]\mathbf{R}_{\mathbf{xx},L}^{-1}[n]\mathbf{r}_L^f[n] + \mathbf{r}_{0L}^f[n+1] \\ &= \mathbf{r}_{0L}^f[n+1] - \mathbf{a}_L^T[n+1]\mathbf{r}_L^f[n] = \alpha_L^f[n+1]. \end{aligned}$$

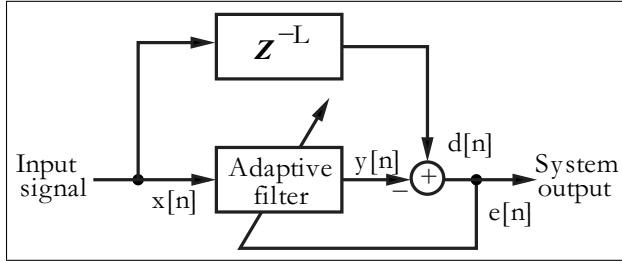


Fig. 8.31. Linear backward prediction of order L

We can now rewrite $\mathbf{R}_{\mathbf{x}\mathbf{x},L+1}^{-1}[n+1]$ from (8.77) as

$$\mathbf{R}_{\mathbf{x}\mathbf{x},L+1}^{-1}[n+1] = \left[\begin{array}{c|c} \frac{1}{\alpha_L^f[n+1]} & \frac{\mathbf{a}_L^T[n+1]}{\alpha_L^f[n+1]} \\ \hline \frac{\mathbf{a}_L[n+1]}{\alpha_L^f[n+1]} & \mathbf{R}_{\mathbf{x}\mathbf{x},L}^{-1}[n] + \frac{\mathbf{a}_L[n+1]\mathbf{a}_L^T[n+1]}{\alpha_L^f[n+1]} \end{array} \right]. \quad (8.80)$$

After some rearrangements (8.77) can be written as

$$\begin{aligned} \mathbf{k}_{L+1}[n+1] &= \left[\begin{array}{c} 0 \\ \mathbf{k}_L[n+1] \end{array} \right] + \frac{\epsilon_L^f[n+1]}{\alpha_L^f[n+1]} \left[\begin{array}{c} 1 \\ \mathbf{a}_L[n+1] \end{array} \right] \\ &= \left[\begin{array}{c} \mathbf{g}_L[n+1] \\ \gamma_L[n+1] \end{array} \right]. \end{aligned}$$

Unfortunately, we do not have a closed recursion so far. For the iterative update of the Kalman gain vector, we need besides the forward prediction coefficients, also the coefficients of the one-step *backward* predictor, whose *a posteriori* error function is

$$\epsilon^b[n] = x[n-L] - \hat{x}[n-L] = x[n-L] - \mathbf{b}^T[n]\mathbf{x}[n], \quad (8.81)$$

again all vectors are of size $(L \times 1)$. The linear backward predictor is shown in Fig. 8.31.

The *a priori* error for the backward predictor is given by

$$\epsilon_L^b[n] = x[n-L] - \mathbf{b}^T[n-1]\mathbf{x}_L[n],$$

The iterative equation to compute the least-square coefficients for the backward predictor is equivalent to the forward case and given by

$$\mathbf{R}_{\mathbf{x}\mathbf{x}}[n]\mathbf{b}[n] = \mathbf{r}^b[n] = \sum_{s=0}^n \mathbf{x}[s]\mathbf{x}[s-L], \quad (8.82)$$

and the minimum value for the total squared error becomes

$$\alpha^f[n] = r_0^b[n] - \mathbf{b}^T[n]\mathbf{r}^b[n],$$

where $r_0^b[n] = \sum_{s=0}^n x[s-L]^2$. To update the backward predictor coefficient we need again the Kalman gain factor in (8.69) as for the updating of the filter coefficients, namely

$$\mathbf{b}_L[n+1] = \mathbf{b}_L[n] + \mathbf{k}_L[n+1]e_L^b[n+1].$$

Now we can again find a Levinson–Durbin type of recursive equation for the extended Kalman gain vector, only this time using the backward prediction coefficients. It follows that:

$$\mathbf{k}_{L+1}[n+1] = \mathbf{R}_{\mathbf{xx},L+1}^{-1}[n+1]\mathbf{x}_{L+1}[n+1]. \quad (8.83)$$

$$= \begin{bmatrix} \mathbf{R}_{\mathbf{xx},L}[n] & | & \mathbf{r}_L^b[n+1] \\ \mathbf{r}_L^{bT}[n] & | & \mathbf{r}_{0L}^b[n+1] \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{x}_L[n+1] \\ \mathbf{x}[n-L+1] \end{bmatrix}. \quad (8.84)$$

To solve the matrix inversion, we define as in (8.79) a $(L+1) \times (L+1)$ block matrix \mathbf{M} , only this time the block \mathbf{A} needs to be nonsingular and it follows that:

$$\mathbf{M}^{-1} = \begin{bmatrix} \mathbf{A} & | & \mathbf{B} \\ \hline \mathbf{C} & | & \mathbf{D} \end{bmatrix}^{-1} = \frac{\mathbf{A}^{-1} - (\mathbf{A}^{-1}\mathbf{B}\mathbf{C}\mathbf{A}^{-1})(\mathbf{C}\mathbf{A}^{-1}\mathbf{B} - \mathbf{D})^{-1}}{(\mathbf{C}\mathbf{A}^{-1}\mathbf{B} - \mathbf{D})^{-1}\mathbf{C}\mathbf{A}^{-1}} \begin{vmatrix} \mathbf{A}^{-1}\mathbf{B}(\mathbf{C}\mathbf{A}^{-1}\mathbf{B} - \mathbf{D})^{-1} \\ -(CA^{-1}B - D)^{-1} \end{vmatrix}.$$

We now make the following associations:

$$\mathbf{A} = \mathbf{R}_{\mathbf{xx},L}[n] \quad \mathbf{B} = \mathbf{r}_L^b[n+1]$$

$$\mathbf{C} = \mathbf{r}_L^{bT}[n] \quad \mathbf{D} = \mathbf{r}_{0L}^b[n+1],$$

we then get the following intermediate results:

$$\mathbf{A}^{-1}\mathbf{B} = \mathbf{R}_{\mathbf{xx},L}^{-1}[n]\mathbf{r}_L^b[n+1] = \mathbf{b}_L[n+1]$$

$$\mathbf{C}\mathbf{A}^{-1} = \mathbf{r}_L^{bT}[n+1]\mathbf{R}_{\mathbf{xx},L}^{-1}[n] = \mathbf{b}_L^T[n+1]$$

$$-(\mathbf{C}\mathbf{A}^{-1}\mathbf{B} - \mathbf{D}) = -\mathbf{b}_L^T[n+1]\mathbf{r}_L^b[n+1] + \mathbf{r}_{0L}^b[n+1] = \alpha_L^b[n+1].$$

Using this intermediate results in (8.81) we get

$$\mathbf{R}_{\mathbf{xx},L+1}^{-1}[n] = \begin{bmatrix} \mathbf{R}_{\mathbf{xx},L}^{-1}[n] + \frac{\mathbf{b}_L[n+1]\mathbf{b}_L^T[n+1]}{\alpha_L^b[n]} & | & \frac{\mathbf{b}_L^T[n+1]}{\alpha_L^b[n]} \\ \hline \frac{\mathbf{b}_L[n+1]}{\alpha_L^b[n]} & | & \frac{1}{\alpha_L^b[n]} \end{bmatrix}.$$

After some rearrangements (8.83) can now, using the backward prediction coefficients, be written as

$$\begin{aligned} \mathbf{k}_{L+1}[n+1] &= \begin{bmatrix} \mathbf{k}_L[n+1] \\ 0 \end{bmatrix} + \frac{\epsilon_L^b[n+1]}{\alpha_L^b[n+1]} \begin{bmatrix} \mathbf{b}_L[n+1] \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{g}_L[n+1] \\ \gamma_L[n+1] \end{bmatrix}. \end{aligned}$$

The only iterative update equation missing so far is for the minimum values of the total square errors, which is given by

$$\alpha_L^f[n+1] = \alpha_L^f[n] + \epsilon_L^f[n+1]e_L^f[n+1] \quad (8.85)$$

$$\alpha_L^b[n+1] = \alpha_L^b[n] + \epsilon_L^b[n+1]e_L^b[n+1]. \quad (8.86)$$

We now have all iterative equations available to define the

Algorithm 8.10: Fast Kalman RLS Algorithm

The prewindowed fast Kalman RLS algorithm to adjust the L filter coefficients of an adaptive filter uses the following steps:

- 1) Initialize $\mathbf{x} = \mathbf{a} = \mathbf{b} = \mathbf{f} = \mathbf{k} = [0, 0, \dots, 0]^T$ and $\alpha^f = \alpha^b = \delta$
- 2) Accept a new pair of input samples $\{x[n+1], d[n+1]\}$.
- 3) Compute now the following equations to update \mathbf{a}, \mathbf{b} , and \mathbf{k} in sequential order:

$$\begin{aligned} e_L^f[n+1] &= x[n+1] - \mathbf{a}^T[n] \mathbf{x}_L[n] \\ \mathbf{a}_L[n+1] &= \mathbf{a}_L[n] + \mathbf{k}_L[n] e_L^f[n+1] \\ \epsilon_L^f[n+1] &= x[n+1] - \mathbf{a}^T[n+1] \mathbf{x}_L[n] \\ \alpha_L^f[n+1] &= \alpha_L^f[n] + \epsilon_L^f[n+1] e_L^f[n+1] \\ \mathbf{k}_{L+1}[n+1] &= \begin{bmatrix} 0 \\ \mathbf{k}_L[n+1] \end{bmatrix} = \frac{\epsilon_L^f[n+1]}{\alpha_L^f[n+1]} \begin{bmatrix} 1 \\ \mathbf{a}_L[n+1] \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{g}_L[n+1] \\ \gamma_L[n+1] \end{bmatrix} \\ e_L^b[n+1] &= x[n+1-L] - \mathbf{b}^T[n] \mathbf{x}_L[n+1] \\ \mathbf{k}_L[n+1] &= \frac{\mathbf{g}_L[n+1] - \gamma_L[n+1] \mathbf{b}^T[n]}{1 + \gamma_L[n+1] e_L^b[n+1]} \\ \mathbf{b}_L[n+1] &= \mathbf{b}_L[n] + \mathbf{k}_L[n+1] e_L^b[n+1]. \end{aligned}$$

- 4) Shift the $x[n+1]$ in the reference signal vector $\mathbf{x}[n+1]$ and compute the following two equations in order to update the adaptive filter coefficients:

$$\begin{aligned} e_L[n+1] &= d[n+1] - \mathbf{f}_L^T[n] \mathbf{x}_L[n+1] \\ \mathbf{f}_L[n+1] &= \mathbf{f}_L[n] + \mathbf{k}_L[n+1] e_L[n+1]. \end{aligned}$$

Next continue with step 2.

Counting the computational effort we find that step 3 needs 2 divisions, $8L+2$ multiplications, and $7L+2$ add or subtract operations. The coefficient update in step 4 uses an additional $2L$ multiply and add/subtract operations, that the total computational effort is $10L+2$ multiplications, $9L+2$ add/subtract operations and 2 divisions.

8.6.3 The Fast a Posteriori Kalman RLS Algorithm

A careful inspection of Algorithm 8.10 reveals that the original fast Kalman algorithm as introduced by Ljung et al. [297] is mainly based on the *a priori* error equations. In the fast a posteriori error sequential technique (FAEST) introduced by Carayannis et al. [298] to a greater extent the *a posteriori* error is used. The algorithm explores even more the iterative nature of the different parameters in the fast Kalman algorithm, which will reduce the computational effort by an additional $2L$ multiplications. Otherwise, the original

fast Kalman and the FAEST use mainly the same ideas, i.e., extended by one length Kalman gain, and the use of the forward and backward predictions \mathbf{a} and \mathbf{b} . We also introduce the forgetting factor ρ . The following listing shows the inner loop of the FAEST algorithm in MATLAB:

```
%***** FAEST Update of k, a, and b
ef=xin - a'*x;           % a priori forward prediction error
ediva=ef/(rho*af);       % a priori forward error/minimal error
ke(1)=-ediva;            % extended Kalman gain vector update
ke(2:l+1)=k - ediva*a;% split the l+1 length vector
epsf=ef*psi;              % a posteriori forward error
a=a+epsf*k;               % update forward coefficients
k=ke(1:l) + ke(l+1).*b;   % Kalman gain vector update
eb=-rho*alphab*ke(l1);    % a priori backward error
alphahf=rho*alphah+ef*epsf; % forward minimal error
alpha=alpha+ke(l+1)*eb+ediva*ef; % prediction crosspower
psi=1.0/alpha;             % psi makes it a 2 div algorithm
epsb=eb*psi;               % a posteriori backward error update
alphab=rho*alphab+eb*epsb; % minimum backward error
b=b-k*epsb;                % update backward prediction coefficients
x=[xin;x(1:l-1)]; % shift new value into filter taps
%***** Time updating of the LS FIR filter
e=din-f'*x;                % error: reference - filter output
eps=-e*psi;                  % a posteriori error of adaptive filter
f=f+w*eps;                   % coefficient update
```

The total effort (not counting the exponential weight with ρ) is 2 divisions, $7L + 8$ multiplications and $7L + 4$ additions or subtractions.

8.7 Comparison of LMS and RLS Parameters

Finally, Table 8.3 compares the algorithms we have introduced in this chapter. The table shows a comparison in terms of computation complexity for the basic stochastic gradient (SG) methods like signed LMS (SLMS), normalized LMS (NLMS) or block LMS (BLMS) algorithm using a FFT. Transform-domain algorithms are listed next, but the effort does not include the power normalization, i.e., L normalizations in the transform domain. From the RLS algorithms we have discussed the (fast) Kalman algorithm and the FAEST algorithm. Lattice algorithm (not discussed) in general, require a large number of division and square root computations and it has been suggested to use the logarithmic number system (see Chap. 2, p. 69) in this case [299].

The data in Table 8.3 are based on the discussion in Chap. 6 of DCT and DFT and their implementation using fast DIF or DIT algorithms. For DCT or DFT of length 8 and 16 more efficient (Winograd-type) algorithms have

Table 8.3. Complexity comparison for LMS and RLS algorithms for length- L adaptive filter. TDLMS without normalization. Add L multiplications and $2L$ add/subtract and L divide, if normalization is used in the TDLMS algorithms

Algorithm	Implementation	Computational load		
		mult	add/sub	div
SG	LMS	$2L$	$2L$	-
	SLMS	L	$2L$	-
	NLMS	$2L + 1$	$2L + 2$	1
	BLMS (FFT)	$10 \log_2(L) + 8$	$15 \log_2(L) + 30$	
TDLMS	Hadamard	$2L$	$4L - 2$	-
	Haar	$2L$	$2L + 2 \log_2(L)$	-
	DCT	$2L + \frac{3L}{2} \log_2(L) + L$	$2L + \frac{3L}{2} \log_2(L)$	-
	DFT	$2L + \frac{3L}{2} \log_2(L)$	$2L + \frac{3L}{2} \log_2(L)$	-
	KLT	$2L + L^2 + L$	$2L + 2L$	-
RLS	direct	$2L^2 + 4L$	$2L^2 + 2L - 2$	2
	fast Kalman	$10L + 2$	$9L + 2$	2
	lattice	$8L$	$8L$	$6L$
	FAEST	$7L + 8$	$7L + 4$	2

been developed using even fewer operations. A length-8 DCT (see Fig. 6.23, p. 464), for instance, uses 12 multiplications and a DCT transform-domain algorithm can then be implemented with $2 \times 8 + 12 = 28$ multiplications, which compares to the FAEST algorithms $7 \times 8 + 8 = 64$. But this calculation does not take into account that a power normalization is mandatory for all TDLMS (otherwise there is no fast convergence compared with the standard LMS algorithm [287, 288]). The effort for the division may be larger than the multiplication effort. When the power normalization factor can be determined beforehand it may be possible to implement the division with hardwired scaling operations. FAEST needs only 2 divisions, independent of the ADF length.

A comparison of the RLS and LMS adaptation speed was presented in Example 8.9 (p. 580), which shows that RLS-type algorithms adapt much faster than the LMS algorithm, but the LMS algorithm can be improved essentially with transform-domain algorithms, like the DCT-LMS. Also, error floor and consistency of the error is, in general, better for the RLS algorithm, when compared with LMS or TDLMS algorithms. But none of the RLS-type algorithms can be implemented without division operations, which will require usually a larger overall system bit width, at least a fractional number representation, or even a floating-point representation [299]. The LMS algorithm on the other hand, can be implemented with only a few bits as presented in Example 8.5 (p. 561).

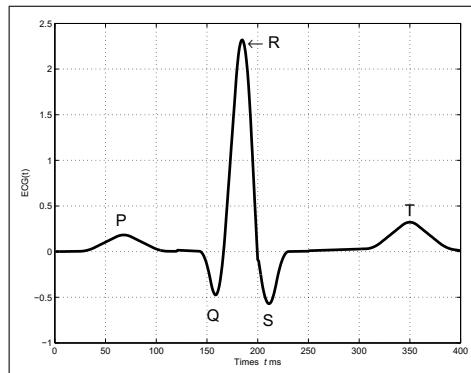


Fig. 8.32. ECG example signal

8.8 Principle Component Analysis (PCA)

The signal processing methods discussed so far only had a single input signal and most model signals were well defined. The tasks we would now like to discuss are more complicated, have multiple inputs and outputs, and the precise signal models are mostly not known. Such a task is for instance the measurement and evaluation of an electrocardiogram (ECG) that measures the heart's electrical activity. We would typically have from 3 to 12 such measurements taken from different location on the body, i.e., 3 bipolar and 3 unipolar between the arms and legs and 6 around the heart. A healthy ECG signal [300] would consist of a regular pulse where each pulse consists of a pre-wave P (ca. 80 ms), a main peak QRS (80 – 120 ms), and a post wave T (160 ms) complex. Typical amplitudes are $P = \leq 0.2$ mV; $Q = R/4$; $R = 0.6 – 2.6$ mV; $S < 0.6$ mV, and $T = R/7$; see Fig. 8.32. The task would be to combine the up to 12 (noisy) signals, into a reliable single signal, or 2 (or 3) if we have to monitor fetal (or twin) heart rates too [301]. For a fetal monitor a typical task would be to make sure that the fetal heart rate stays in the desired range between 110 beats per minute (bpm) and 180 bpm.

Although quite old, the method of choice to reduce the dimensions of the input space is the Kahunen–Loevé transform, a.k.a *principle component analysis* (PCA). To implement PCA we build the cross correlation matrix between our input signals, compute the eigenvalues and eigenvectors, and then transform the input signals using the eigenvectors and eigenvalues. Putting this into equations gives

$$\mathbf{C}_{\mathbf{x}\mathbf{x}} = E\{\mathbf{x} - \boldsymbol{\eta}\}^T(\mathbf{x} - \boldsymbol{\eta}) \quad (8.87)$$

Let \mathbf{e}_k be the eigenvector number k and λ_k the corresponding eigenvalue of $\mathbf{C}_{\mathbf{x}\mathbf{x}}$. Then we get the k 's principle component using

$$y_k = \mathbf{e}_k^T \mathbf{x} \quad (8.88)$$

The spread of the eigenvalues is a good indicator that “how many” principle components are in our input data and we usually drop the signals associated with small eigenvalues. As a result we will have one or more signals that represent the input signals in an minimum mean square sense. The principle components capture the power of the majority of the signals and at the same time all output signals are orthogonal to each other. That is why PCA sometimes is also called a *whitening* operation. Now let us build a fetal monitoring ECG model.

Example 8.11: Dimension Reduction in ECG Data

Let us use a simplified ECG model where the mother and fetus ECG are modeled as impulse trains. The sampling frequency is set to 1 kHz and, with a period length of 14, the mother ECG should be constant at 71 bpm and the fetal period length should be 8 or 125 bpm. Let us assume that we used four channel measurements that give a superposition of the two signals. The polarity of the signals may also be altered. The combination of the inputs is modeled as a random mixing matrix and we assume a 10:1 amplitude scale between mother and fetus ECG amplitudes. The following MATLAB script implements the complete monitoring system:

```
%%%%%% MatLab script ECG PCA example
clear all; close all;
T1=14; %% Mother period length at 1 kHz
T2=8; %% Fetus period length at 1 kHz
T=T1*T2*2; %% Total samples in simulation
s=zeros(2,T);
s(1,:)=upsample(ones(1,T/T1),T1,T1/2); %% Mother ECG
s(2,:)=upsample(ones(1,T/T2),T2,T2/2)/10; %% Fetus ECG
subplot(3,1,1);L=50;t=1:L; %% Plot signals without noise
plot(t,s(1,1:L),'k-x',t,s(2,1:L),'k-o');
title('Undisturbed mother (x) and fetus (o) signals');
axis([0 L -.2 1.2]);ylabel('s[n]')
rng(0); %% Initialize random number to start
A=[-1 -1;-1 1;1 -1;1 1];
A=A+(rand(4,2)-.5); %% Mixing matrix with polarity change
x=A*s+rand(4,T)/100; %% Apply mixing and add Gauss noise
subplot(3,1,2); %% Plot measured signals with noise
plot(t,x(1,1:L),'k-x',t,x(2,1:L),'k-o',...
    t,x(3,1:L),'k-+',t,x(4,1:L),'k-*');
title('The 4 measured ECG signals (with noise)')
axis([0 L -1 2]);ylabel('x[n]')
Cxx = cov(x', 1) % Calculate the covariance matrix
disp('Eigenvectors and eigenvalues of covariance matrix:')
[E, D] = eig(Cxx)
%apply reconstruction mixing to the x data i.e. do PCA
y=(D.^-.5 * E'*x);
subplot(3,1,3);
plot(t,y(4,1:L),'k-x',t,y(3,1:L),'k-o');
title('Mother (x) and fetus (o) signals after PCA')
```

```

grid
axis([0 L -.5 4.5]); xlabel('Sample n'); ylabel('y[n]')
print -deps ecg1.eps; print -djpeg ecg1.jpg

```

As can be seen first the two test signals are generated and plotted. Then the mixing matrix is applied to produce four measurements and Gaussian noise is added to the signals. Then the PCA is performed using the cross-correlation matrix C_{xx} . The values of the cross-correlation matrix provided by MATLAB are

0.0310	0.0264	-0.0276	-0.0635
0.0264	0.0244	-0.0264	-0.0551
-0.0276	-0.0264	0.0291	0.0583
-0.0635	-0.0551	0.0583	0.1308

Finally the reconstruction of the two signals is performed. Note that MATLAB gives out the eigenvalues as diagonal elements in square matrix and sorts the eigenvalues from small to large, i.e., D becomes

0.0000	0	0	0
0	0.0000	0	0
0	0	0.0037	0
0	0	0	0.2115

and the associate eigenvector matrix E

-0.6078	-0.5971	-0.3604	-0.3798
0.5737	-0.6215	0.4152	-0.3350
0.4797	-0.2568	-0.7594	0.3567
-0.2672	-0.4374	0.3479	0.7850

As expected we get only two nonzero eigenvalues and therefore only two principle components.

The simulation results in Fig. 8.33 show first the two inputs signals, followed by the four measured signals. The third plot shows the two reconstructed ECG signals. Note that the period of the signals, i.e., 8 and 14, is well preserved; only the amplitude comes out with a different scaling than the original data. PCA has done the dimension reduction from four to two signals and at the same time a signal separation.

8.11

8.8.1 Principle Component Analysis Computation

The computation of the PCA requires a substantial amount of arithmetic. The cross correlation matrix can be iteratively compute via

$$\mathbf{C}_{xx}(t+1) = \beta E\{x(t)^T x(t)\} + (1 - \beta) \mathbf{C}_{xx}(t). \quad (8.89)$$

Depending on the noise level this may take many clock cycles. For our low noise two input example after 50 – 100 samples we should have sufficient stable values. The computation can be simplified a little, since the correlation matrix \mathbf{C}_{xx} is symmetric. This will also simplify the eigenvalues and eigenvectors since for a symmetric matrix the eigenvalues and eigenvectors are real and the eigenvectors are all orthogonal, and the eigenvector matrix becomes unitary, i.e., $\mathbf{H}^T = \mathbf{H}^{-1}$. The computation of the eigenvalues and eigenvectors however is substantially more complicated than the computation

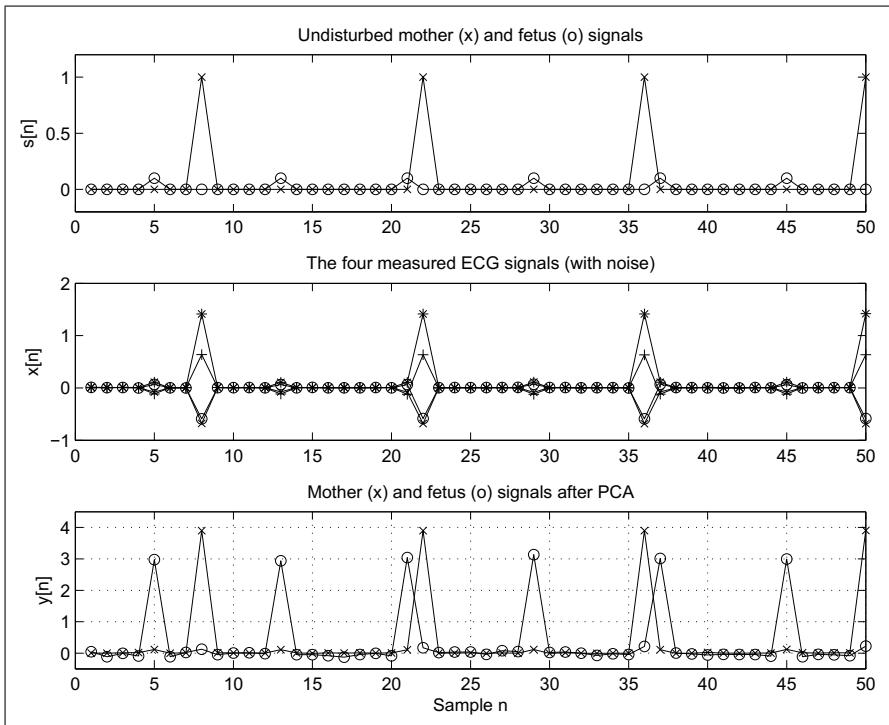


Fig. 8.33. Simulation of PCA ECG example

of the cross correlation matrix. From the literature [302, 303] we know there are three major categories:

- Direct method
- Power method after v. Mises
- Neural learning methods

The methods differ substantially in their hardware requirement, coding of algorithm, number of iterations needed for convergence, and special cases to be considered. Let us have a closer look at each of these methods next.

In the *direct method* to compute the eigenvalues and eigenvectors we first determine the eigenvalues by computing the characteristic polynomial, i.e., $P(\lambda) = \det(\mathbf{C}_{xx} - \mathbf{I}\lambda)$. The zeros of the polynomial are the eigenvalues λ_k . To compute the eigenvectors we need to solve the matrix system $\mathbf{C}_{xx}\mathbf{e}_k = \lambda_k\mathbf{e}_k$. It can be seen that the algorithm does not require any iterations or exceptions, and needs just a few matrix equations that can be computed nicely with a microprocessor or MATLAB. However, these matrix systems will translate into a substantial number of hardware resources even for small number of signals [304].

The second algorithm that just needs a few iterations called *power method* has been credited to v. Mises [302, 303]. It computes the sequence of vector $\mathbf{v}^{(k)}$:

$$\begin{aligned}\mathbf{v}^{(2)} &= \mathbf{C}_{xx}\mathbf{v}^{(1)} \\ \mathbf{v}^{(3)} &= \mathbf{C}_{xx}\mathbf{v}^{(2)} = \mathbf{C}_{xx}^2\mathbf{v}^{(1)} \\ \mathbf{v}^{(4)} &= \mathbf{C}_{xx}\mathbf{v}^{(3)} = \mathbf{C}_{xx}^3\mathbf{v}^{(1)} \dots\end{aligned}$$

where $\mathbf{v}^{(1)}$ is random starting vector and the superscript (k) indicates the iteration. The power method does not require one to compute the power of the correlation matrix, since

$$\mathbf{C}_{xx}^{k+1}\mathbf{v} = \mathbf{C}_{xx}(\mathbf{C}_{xx}^k\mathbf{v}). \quad (8.90)$$

It can be shown that with proper initialization the vector converges to the dominant eigenvector $\mathbf{v}^{(k)} \rightarrow \mathbf{e}_1$. The only exception is this does not work if $\mathbf{v}^{(1)}$ is already another eigenvector like \mathbf{e}_2 . The convergence of this method is quite fast as the following short example shows.

Example 8.12: The 4×4 correlation matrix from Example 8.11 give the following v. Mises iteration when using an initial vector $\mathbf{v}^{(1)} = [11\dots]^T$:

$$\begin{aligned}\mathbf{v}^{(2)} &= \mathbf{C}_{xx}\mathbf{v}^{(1)} = [-0.033814 \quad -0.030796 \quad 0.033233 \quad 0.070448]^T \\ \mathbf{v}^{(3)} &= \mathbf{C}_{xx}\mathbf{v}^{(2)} = [-0.007254 \quad -0.006402 \quad 0.006819 \quad 0.014996]^T \\ \mathbf{v}^{(4)} &= \mathbf{C}_{xx}\mathbf{v}^{(3)} = [-0.001535 \quad -0.001354 \quad 0.001442 \quad 0.003172]^T\end{aligned}$$

Finally we normalize the vector to length one via $\mathbf{v}=\mathbf{v}./\text{sqrt}(\text{sum}(\mathbf{v}.^2))'$, and get

$$\mathbf{e}_1 = [-0.379796 \quad -0.334976 \quad 0.356721 \quad 0.785046]^T \quad (8.91)$$

which matches the exact eigenvector for six fractional digits. The error of the approximation $\log_2(|\mathbf{E}(:, 4) - \mathbf{v}^{(k)}|)$ increases from 5.1, 10.9 to 16.8 bits after only three iterations.

The eigenvalue can be computed via $\lambda_1 = \mathbf{v}^{(4)}(i)/\mathbf{v}^{(3)}(i) = 0.211573$ that has four digit accuracy.

8.12

The next eigenvalues can be computed in a similar way since we know that all eigenvectors are orthogonal. In the method outlined by v. Mises we use a random vector \mathbf{r} and compute $c_1 = \mathbf{r}^T \mathbf{e}_1$. Then we initialize our power matrix algorithm with $\mathbf{u}^{(1)} = \mathbf{r} - c_1 \mathbf{e}_1$ and proceed as usual.

Example 8.13: The 4×4 correlation matrix from Example 8.12 produces \mathbf{e}_1 . We now use an initial vector $\mathbf{r} = [11\dots]^T$ and get $c_1 = \mathbf{r}^T \mathbf{e}_1 = 0.4270$; the iteration then starts with the vector $\mathbf{u}^{(1)} = \mathbf{r} - c_1 \mathbf{e}_1 = [1.1622 \quad 1.1430 \quad 0.8477 \quad 0.6648]^T$. The algorithm for the second eigenvector then proceeds as follows:

$$\begin{aligned}\mathbf{u}^{(2)} &= \mathbf{C}_{xx}\mathbf{u}^{(1)} \\ &= [0.000488 \quad -0.000542 \quad 0.001015 \quad -0.000456]^T \\ \mathbf{u}^{(3)} &= \mathbf{C}_{xx}\mathbf{u}^{(2)}\end{aligned}$$

$$\begin{aligned}
&= [1.786940 \cdot 10^{-6} \quad -2.058812 \cdot 10^{-6} \quad 3.765611 \cdot 10^{-6} \quad -1.725061 \cdot 10^{-6}]^T \\
\mathbf{u}^{(4)} &= \mathbf{C}_{xx} \mathbf{u}^{(3)} \\
&= [6.662793 \cdot 10^{-9} \quad -7.677155 \cdot 10^{-9} \quad 1.404094 \cdot 10^{-8} \quad -6.432564 \cdot 10^{-9}]^T
\end{aligned}$$

We have 5.4, 14.2, and 23.0 bits precision of $\mathbf{u}^{(k)} \approx \mathbf{e}_2$. The true eigenvector \mathbf{e}_2 needs a final normalization to length one which gives $\mathbf{u}^{(4)}/|\mathbf{u}^{(4)}|$:

$$\mathbf{e}_2 = [0.360358 \quad -0.415220 \quad 0.759405 \quad -0.347906]^T \quad (8.92)$$

and all six decimal digits come out correct.

The eigenvalue can be computed via $\lambda_2 = \mathbf{u}^{(4)}(i)/\mathbf{u}^{(3)}(i) = 0.003729$.

8.13

The advantages of the power method are now apparent. The algorithm needs just a few iterations and only one matrix equation need to be implemented in hardware. The only major drawback of the algorithm comes from the initialization vector. If we by “accident” initialize with an eigenvector belonging to a smaller eigenvalue then the first run will not produce the largest eigenvalue and the related eigenvector. The final normalization of the eigenvector also has the potential to be costly since a vector squaring, square root, and division are needed. It is recommended just to normalize by the largest component of the vector since the multiplication by the eigenvalue will scale the eigenvector anyway. FPGA implementation of the power method will be based on efficient matrix operation implementations such as QR decomposition [305].

The third method we wish to discuss to compute the eigenvectors is the neural network-like learning method that gradually adjusts the vector coefficients similar to how the adaptive filter LMS algorithm works. The basic algorithm uses the Hebbian learning algorithm that adds small increments to the weights proportional to the product between input and output of the network, $\Delta \mathbf{w}_k \approx \mu y_k \mathbf{x}_k$ with $y = \mathbf{w}^T \mathbf{x}$. Oja and Karhunen [306] showed that \mathbf{w}_k will converge to the eigenvector of the system. The algorithm however needs proper normalization to avoid growth over all limits and this results in the equation below known as the Oja learning rule [306] shown here for the first eigenvector

$$\Delta \mathbf{w}_1 = \mu y_1 (\mathbf{x} - y_1 \mathbf{w}_1) \quad (8.93)$$

which is usually called a stochastic gradient ascent (SGA) algorithm. Sanger in his Master’s thesis later made a small correction to the method that makes it easier to implement [307]. The Sanger method can be summarized as

$$\Delta \mathbf{w}_k = \mu y_k \left(\mathbf{x} - \sum_{i=1}^k y_i \mathbf{w}_i \right). \quad (8.94)$$

This method is named the generalized Hebbian algorithm (GHA). For a two channel system Fig. 8.34 shows the first and Fig. 8.35 the second eigenvector as a SIMULINK model. The outputs y_k are the desired signals from the PCA analysis.

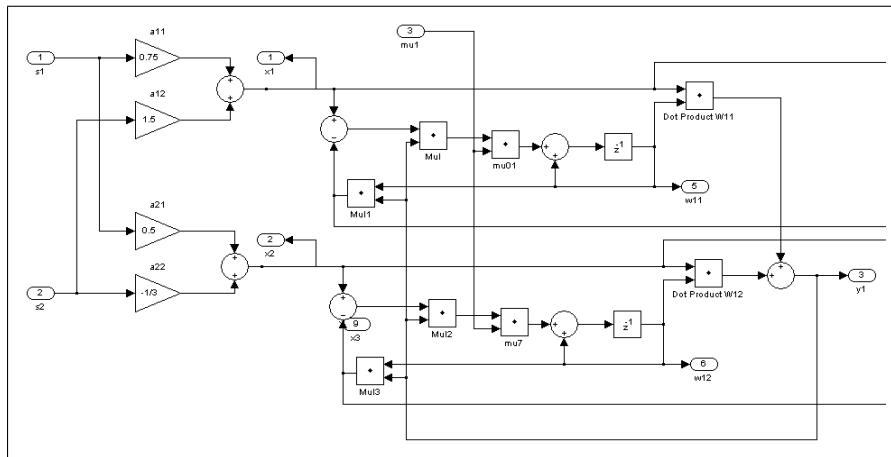


Fig. 8.34. Mixing matrix and first principle component learning system using Sanger's GHA algorithm

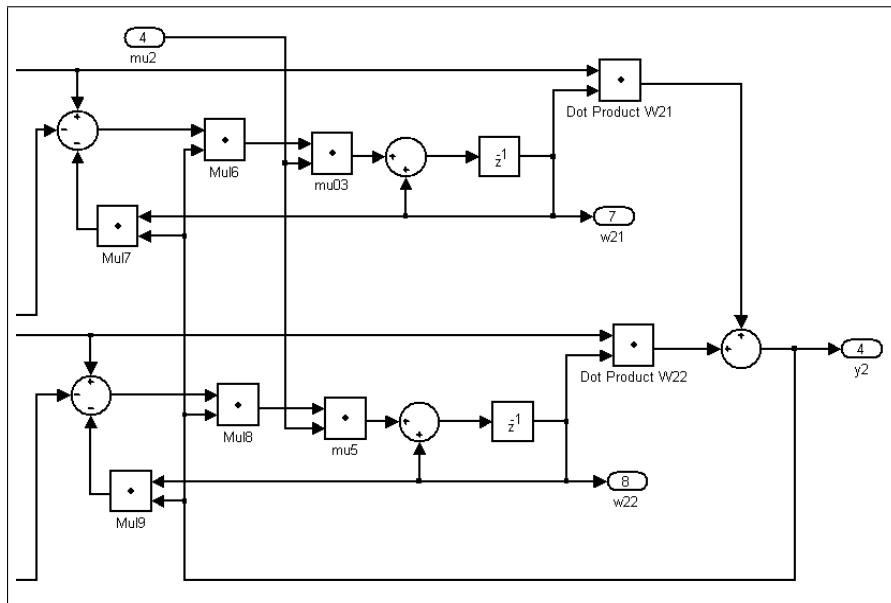


Fig. 8.35. Second principle component learning system using Sanger's GHA algorithm

8.8.2 Implementation of Sanger's GHA PCA

Since the GHA seemed to be the best in terms of required hardware resources [308] we would like to discuss a design example in the following.

Example 8.14: PCA Design Using Sanger's GHA

Let us assume we have a binary digital signal $s_1 = \pm 0.25\text{rect}(t/5)$ and a sine wave as inputs $s_2 = \sin(2\pi t/6)$. The sine wave has a period six and the digital signals five samples per bit. These two signals are then combined with a mixing matrix

$$\mathbf{x} = \begin{bmatrix} 3/4 & 3/2 \\ 1/2 & -1/3 \end{bmatrix} s \quad (8.95)$$

to produce the two system inputs x_1 and x_2 ; see Fig. 8.34. After an initial learning phase we would expect that the PCA system will converge and separate our initial signals. Convergence is achieved after 4K clock cycles. Since two eigenvectors need to be learned we first start with e_1 . After 1000 clock cycles we gradually reduce the learning rate to zero and then start with learning of the second eigenvector. After another 1000 clock cycles we then reduce the learning rate to zero to have a stable output vector e_2 . The learning rates μ and the eigenvector values are shown in Fig. 8.36a. If we compute the cross-correlation matrix and the associate eigenvector we get

$$\mathbf{C}_{xx} = \begin{bmatrix} 1.1599 & -0.2265 \\ -0.2265 & 0.0712 \end{bmatrix} \quad (8.96)$$

$$\mathbf{e}_1 = [-0.9806 \quad 0.1959] \quad \lambda_1 = 1.2051 \quad (8.97)$$

$$\mathbf{e}_2 = [-0.1959 \quad -0.9806] \quad \lambda_2 = 0.0259 \quad (8.98)$$

If we compare this data with the GHA we see a sign change in both eigenvectors but otherwise a match in the values.

The simulation of the GHA is shown in Fig. 8.36b starting with sample 4000. At this point in the simulation we have stable eigenvectors. We see the two input signals s_k , followed by the mixing output x_k and the output signals y_k . The input signals are well recovered.

The VHDL design⁵ for a PCA design using Sanger's generalized Hebbian algorithm is shown in the following listing:

```

LIBRARY ieee; USE ieee.std_logic_1164.ALL;
PACKAGE n_bit_int IS
  -- User defined types
  SUBTYPE SLV32 IS STD_LOGIC_VECTOR(31 DOWNTO 0);
END n_bit_int;
LIBRARY work; USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL; USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

LIBRARY ieee_proposed;
USE ieee_proposed.fixed_float_types.ALL;
USE ieee_proposed.fixed_pkg.ALL;
USE ieee_proposed.float_pkg.ALL;
-----
ENTITY pca IS
  -----> Interface
  PORT (clk      : IN STD_LOGIC; -- System clock
        reset    : IN STD_LOGIC; -- System reset
        s1_in    : IN SLV32;    -- 1. signal input
        s2_in    : IN SLV32;    -- 2. signal input

```

⁵ The equivalent Verilog code `pca.v` for this example can be found in Appendix A on page 864. Synthesis results are shown in Appendix B on page 881.

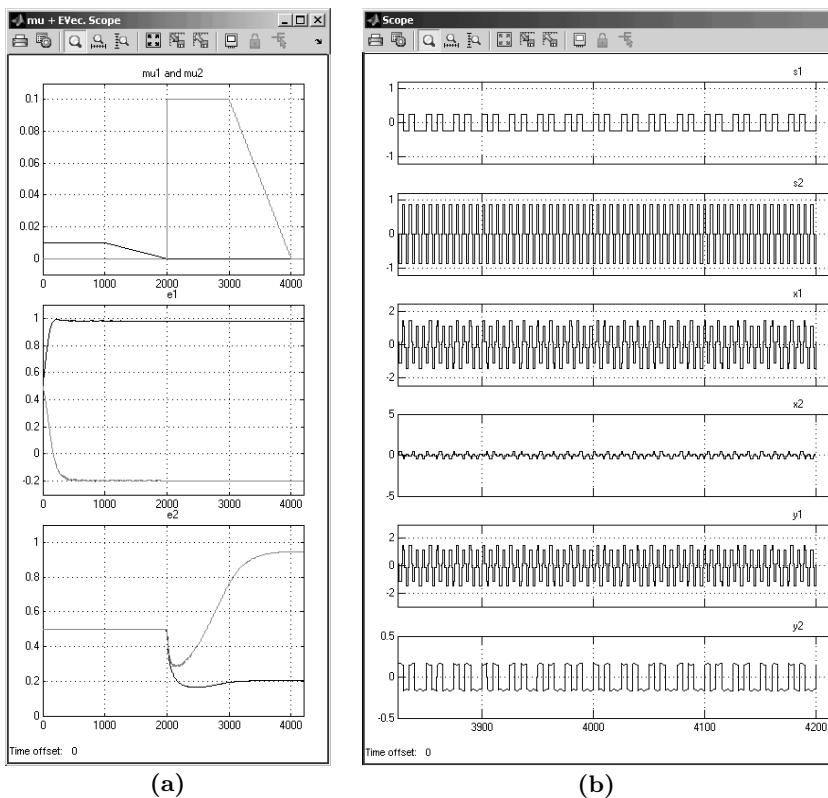


Fig. 8.36. Sanger's GHA simulation in SIMULINK. (a) Overall learning rate and eigenvectors. (b) Input, mixing, and output signals after convergence

```

mu1_in : IN SLV32;      -- Learning rate 1. PC
mu2_in : IN SLV32;      -- Learning rate 2. PC
x1_out : OUT SLV32;     -- Mixing 1. output
x2_out : OUT SLV32;     -- Mixing 2. output
w11_out : OUT SLV32;    -- Eigenvector [1,1]
w12_out : OUT SLV32;    -- Eigenvector [1,2]
w21_out : OUT SLV32;    -- Eigenvector [2,1]
w22_out : OUT SLV32;    -- Eigenvector [2,2]
y1_out : OUT SLV32;     -- 1. PC output
y2_out : OUT SLV32);    -- 2. PC output
END;
-----
ARCHITECTURE fpga OF pca IS
  CONSTANT a11 : SFIXED(15 DOWNTO -16) :=
                TO_SFIXED(0.75, 15,-16);
  CONSTANT a12 : SFIXED(15 DOWNTO -16) :=
                TO_SFIXED(1.5, 15,-16);
  CONSTANT a21 : SFIXED(15 DOWNTO -16) :=

```

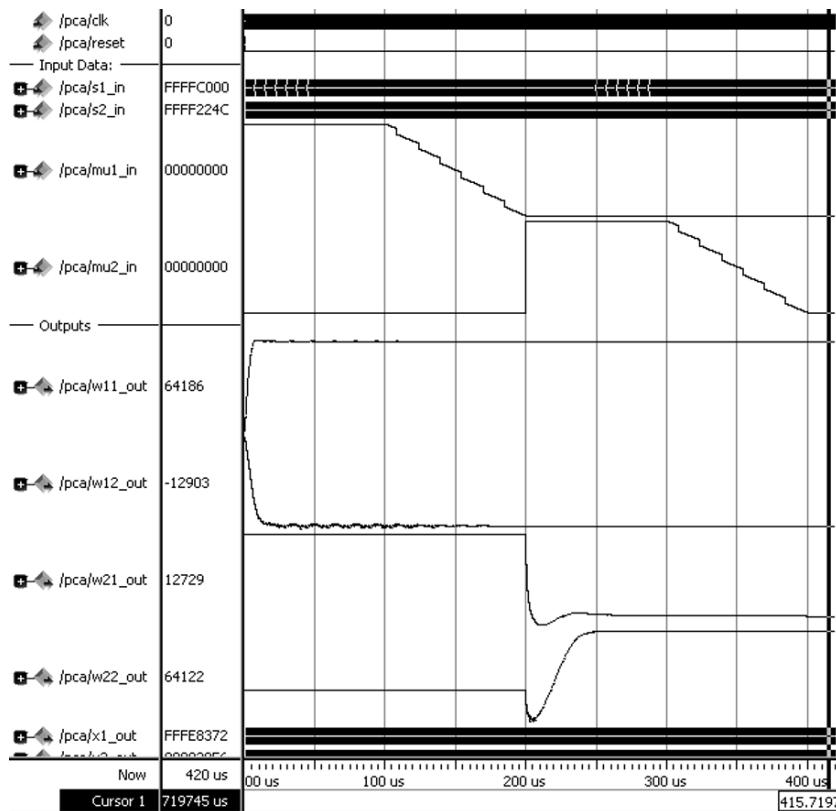


Fig. 8.37. Overall Sanger's GHA simulation in HDL. Learning rate and eigenvectors

```

TO_SFIXED(0.5, 15,-16);
CONSTANT a22 : SFIXED(15 DOWNTO -16) :=
TO_SFIXED(0.333333, 15,-16);
CONSTANT ini : SFIXED(15 DOWNTO -16) :=
TO_SFIXED(0.5, 15,-16);
SIGNAL s, s1, s2, x1, x2, w11, w12, w21, w22, mu1, mu2:
SFIXED(15 DOWNTO -16) := (OTHERS => '0');
BEGIN
  s1 <= TO_SFIXED(s1_in, s); -- Redefine bits as signed
  s2 <= TO_SFIXED(s2_in, s); -- FIX 16.16 number
  mu1 <= TO_SFIXED(mu1_in, s);
  mu2 <= TO_SFIXED(mu2_in, s);

  P1: PROCESS (reset, clk, s1, s2)
  VARIABLE h11, h12, y1, y2 :
  SFIXED(15 DOWNTO -16) := (OTHERS => '0');
  -----> Behavioral Style

```

```

BEGIN
  IF reset = '1' THEN -- Reset/initialize all registers
    x1 <= (OTHERS => '0'); x2 <= (OTHERS => '0');
    w11 <= ini; w12 <= ini;
    w21 <= ini; w22 <= ini;
  ELSIF rising_edge(clk) THEN -- PCA using Sanger GHA
    -- Using the "do not WRAP"
    -- Mixing matrix
    x1<=resize(a11*s1+a12*s2,s,fixed_wrap,fixed_truncate);
    x2<=resize(a21*s1-a22*s2,s,fixed_wrap,fixed_truncate);
    -- First PC and eigenvector
    y1:=resize(x1*w11+x2*w12,s,fixed_wrap,fixed_truncate);
    h11 := resize(w11*y1,s,fixed_wrap,fixed_truncate);
    w11 <= resize(w11+mu1*(x1-h11)*y1,s,
                  fixed_wrap,fixed_truncate);
    h12 := resize(w12*y1,s,fixed_wrap,fixed_truncate);
    w12 <= resize(w12+mu1*(x2-h12)*y1,s,
                  fixed_wrap,fixed_truncate);
    -- Second PC and eigenvector
    y2:=resize(x1*w21+x2*w22,s,fixed_wrap,fixed_truncate);
    w21 <= resize(w21+mu2*(x1-h11-w21*y2)*y2,s,
                  fixed_wrap,fixed_truncate);
    w22 <= resize(w22+mu2*(x2-h12-w22*y2)*y2,s,
                  fixed_wrap,fixed_truncate);
    -- Register y output
    y1_out <= to_slv(y1);
    y2_out <= to_slv(y2);
  END IF;
END PROCESS;

-- Redefine bits as 32 bit SLV
x1_out <= to_slv(x1);
x2_out <= to_slv(x2);
w11_out <= to_slv(w11);
w12_out <= to_slv(w12);
w21_out <= to_slv(w21);
w22_out <= to_slv(w22);

END fpga;

```

The design uses the scheme found in Fig. 8.34 for the first principle component and Fig. 8.35 for the second principle component. The design uses a 32-bit data format and internal a 16.16 signed fractional data type. After specifying the I/O ports the mixing matrix $a_{k,i}$ is defined as constant values. In the architecture first the input signals are redefined in **sfixed** format that should not need any resources. The PCA algorithm is implemented with a single **PROCESS** statement. The register x_k are set to zero and the four weight registers w_k are initialized with a random, nonzero values. Here 0.5 was chosen. After the **rising_edge** statement the mixing is implemented, followed by the code for the first principle component computation and then the second principle component architecture. The **PROCESS** also includes the output assignments to y_k since we like to store these in registers to measure the registered performance of the design. Finally some more internal data are assigned to output pins for monitoring.

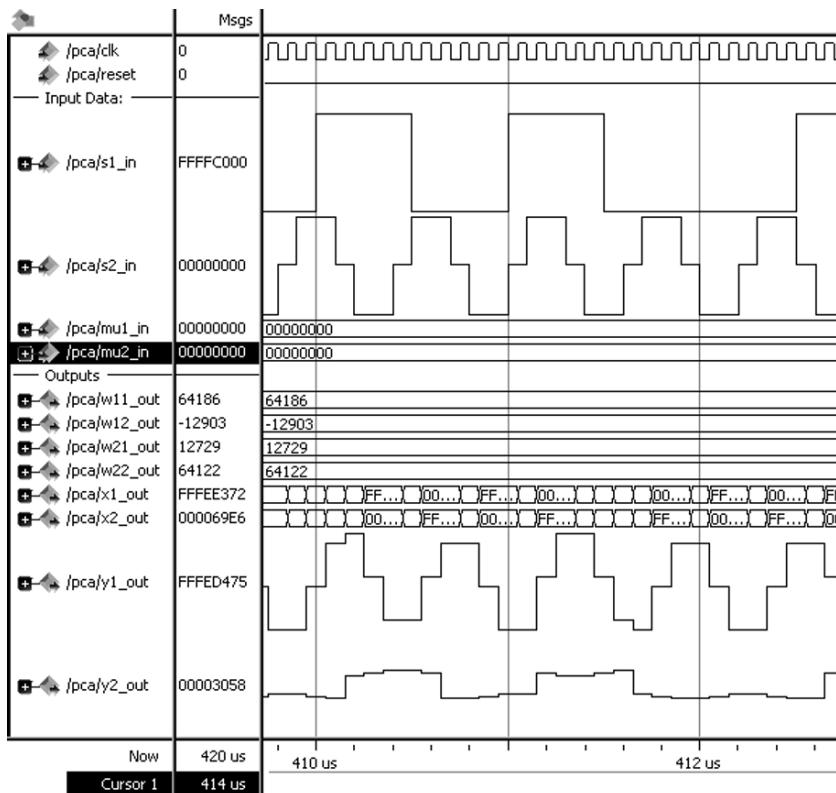


Fig. 8.38. Sanger's GHA simulation in HDL. Input signals and output principle components after convergence

The design uses 2447 LEs, 180 embedded multipliers, and has a registered performance of $F_{max}=18.46$ MHz using the TimeQuest slow 85C model.

The simulation of the GHA from Fig. 8.36 is now reproduced in HDL. First in Fig. 8.37 we see the learning rates and the eigenvectors. Figure 8.38 then shows the input signals and the recovered principle components. At this point in the simulation we have stable eigenvectors.

8.14

The GHA is indeed the slowest PCA algorithm we have discussed so far and we find in the literature several suggestions for improvements. Unfortunately some of the fastest algorithm such as APEX, PAST, or OPAST algorithms [309–311] are *subspace methods* and not PCA that will not compute eigenvectors but a set of rotated basis vectors. Like the LMS algorithm, most of these algorithms replace the slow learning rate with an RLS type version, but in general require substantial matrix computations and also a general division operation that will slow down the hardware registered performance.

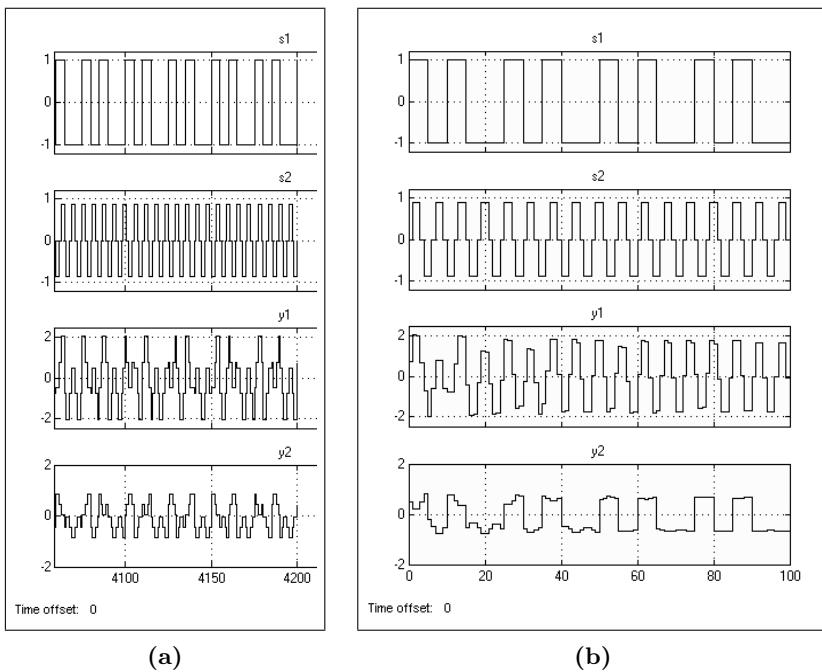


Fig. 8.39. Simulation in SIMULINK with both signals amplitudes one. (a) Using the PCA method. (b) Using the Herault and Jutten ICA method

8.9 Independent Component Analysis (ICA)

The PCA method discussed in the last section works fine to reduce the dimension of the input data space. However, if we use the PCA to separate input signals into independent components this will not always work since the PCA captures the component with maximum power in a mean square sense. Sometime this gives the component we expect, but under certain conditions this does not work. A typical task is blind source separation (BSS) where we try to find the independent sources of a mixture. The term “blind” does not mean we could not measure it, it just mean that we do not know exactly the parameters, shapes or period of the source as in the ECG example from the last section where the exact period is not known a priori. Sometimes just a small change in the system already is sufficient for the PCA method not to produce the desired results. This happens for instance when the signal power of the source are very close, or in the GHA example when the input to the mixing matrix is switched or the amplitude of the digital signal is increased to one. Figure 8.39a shows the output simulation result when the digital signal has amplitude one. The sine and the digital signal are no longer clearly separated.

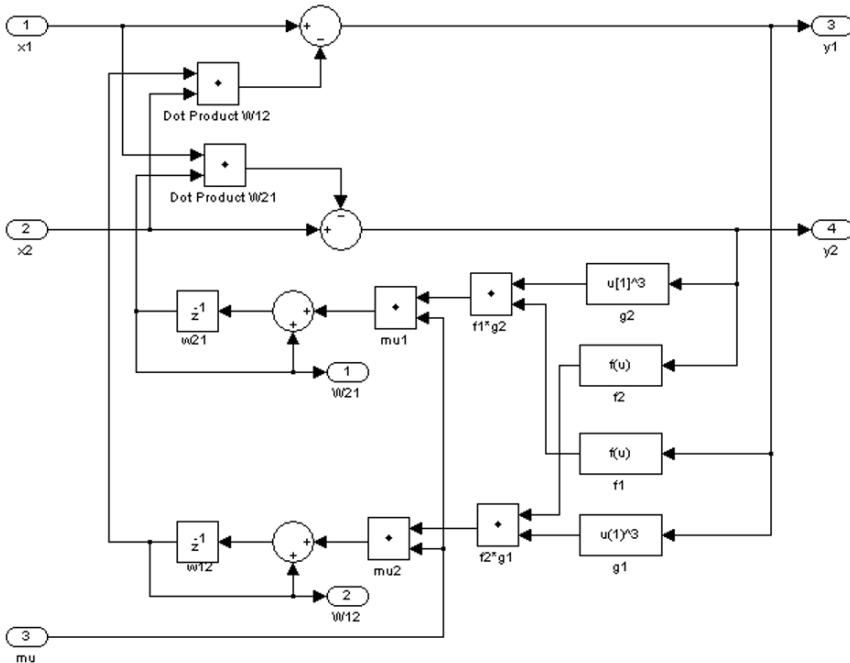


Fig. 8.40. A two channel Herault and Jutten system

One pioneering work that offers a solution to this kind of problem was given by Herault and Jutten [312]. These methods are summarized under the name *Independent Component Analysis* (ICA). They observed that by using higher order statistics the BSS problem can be solved for many tasks that did not work with the PCA method. The higher order statistics are generated via nonlinearities. A summary of typical nonlinearities used in ICA is shown in Fig. 8.41. A two channel system is shown in Fig. 8.40 and the two channel system equations are

$$y_1 = x_1 - x_2 w_{12}$$

$$y_2 = x_2 - x_1 w_{21}$$

$$\Delta w_{21} = \mu f(y_1)g(y_2)$$

$$\Delta w_{12} = \mu f(y_2)g(y_1)$$

where $f()$ and $g()$ are nonlinear functions. One function was chosen to be such as $g(y) = y^3$, while the other was $f(y) = \text{atan}(y)$. They also found that $f(y) = y$ or $f(y) = \text{sign}(y)$ also allowed a separation of the sources. Figure 8.39b shows the two channel experiment with a digital and sine when both amplitudes are one. We notice a short learning time and good separation of

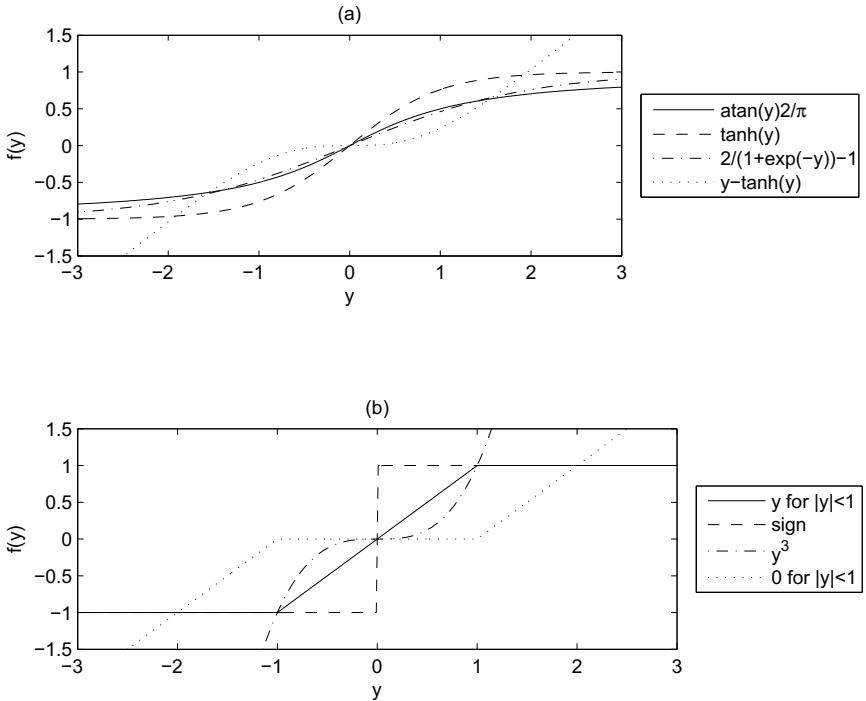


Fig. 8.41. Typical nonlinearities used in ICA. (a) Full precision functions. (b) Reduced complexity approximations

the sources after about 50 clock cycles. The learning rate was 1/16 in the first 50 clock cycles and then reduced to zero in the next 50 clock cycles.

While Herault and Jutten showed some success it was soon discovered that their method had some limitations too. First it cannot be generalized to more than two channels and second it is also very gain dependent since the x_k are directly combined to form the y_k . Several modifications to the ICA had been published in the following years and the results have been summarized in text books [313,314], tutorials [315], and toolboxes. Many ICA methods are closely related. In fact it has been shown that the three theories: minimization of the mutual information, maximization of non-Gaussianity, and maximization of likelihood have the same solutions [316]. Let us briefly review the most popular ICA algorithms in the following.

8.9.1 Whitening and Orthogonalization

Some of the ICA algorithms may need as a preprocessing step the whitening of the signals, i.e., a transformation that the resulting cross-correlation matrix C_{xx} only has nonzero diagonal elements. The PCA method

```
Cxx = cov(x', 1);
[E, D] = eig(covarianceMatrix);
V=D^-.5 * E';
z=V*x;
```

can be used to whiten the signals and the correlation matrix \mathbf{C}_{zz} of \mathbf{z} will have only values zero as non-diagonal elements.

The PCA is just one option for the whitening, but, as we have seen in the last section, already requires substantial resources. There are several methods that seemed to be easier to implement than the PCA if just the whitening is needed. This can be done with an online learning [313, 317] that is similar to the power method, i.e., we have the two matrix equation systems

$$\mathbf{z} = \mathbf{V}\mathbf{x} \quad (8.99)$$

$$\Delta\mathbf{V} = \mu (\mathbf{I} - \mathbf{z}\mathbf{z}^T) \mathbf{V}. \quad (8.100)$$

Another attractive method is the classic orthogonalization method from Gram–Schmidt [302, p. 176] that also produces a white cross-correlation matrix. The method below shows the MATLAB code for three input signals x_k and orthogonal output z_k :

```
% Step K=1
z1 = x1 ./ sqrt(x1'*x1 );
z2 = x2 - (z1'*x2)*z1;
v3 = x3 -(z1'*x3)*z1;
% Step K=2
z2 = z2 ./ sqrt(z2'*z2 );
z3 = z3 - (z2'*z3)*z2;
% Step K=3
z3 = z3 ./ sqrt(z3'*z3 );
```

For the Gram–Schmidt in text books a final normalization is usually carried out; however, if we just need orthogonal and no orthonormal signals this seemed to be unnecessary.

8.9.2 Independent Component Analysis Algorithm

Two classic ICA algorithms are direct extensions of the PCA using nonlinearities. This will lead to the *nonlinear PCA* algorithm that can be implemented in MATLAB as follows:

```
yk = W' * z(:,k);
H = f(yk,n1) * (z(:,k)' - f(yk',n1) * W) ;
W = W + mu * H;
```

where the same nonlinearity $f(yk, n1)$ is used twice, and the input signals are (whitened) row vectors for convenience. The learning rate μ is typically small, e.g., $\mu = 2^{-9}$.

A similar algorithm that also first needs the whitening of the inputs is called the *natural gradient algorithm* (NGA) and can be implemented as

```
yk = W * z(:,k);
H = I - f(yk,n1) * yk';
W = W + mu * H * W;
```

where $I=\text{eye}(N)$ is the identity matrix with one in the diagonal and zero otherwise. This NGA seemed to be a reduced complexity when compared to the nonlinear PCA.

The algorithms discussed so far need a whitening operation first before the actual ICA algorithm, which may be costly. It was shown first by Cardoso and Laheld [317] that we can combine ICA higher order moments processing via nonlinearities with the whitening. They called their algorithm *equivariant adaptive separation via independence* (EASI) and the algorithm works as follows:

```
yk = B * x(:,k);
H = I - yk * yk' + f(yk,n1)*yk' - yk*f(yk,n1)';
B = B + mu * H * B;
```

Note that we now use $x(:,k)$ and not the whitened $z(:,k)$. As nonlinearity Cardoso and Laheld used the `tanh` function [317].

8.9.3 Implementation of the EASI ICA Algorithm

The EASI algorithms seemed to be the most attractive ICA algorithm to be implemented in hardware [317–320] and we wish to design a 2×2 EASI system to separate the sine and digital signal we used in Example 8.14, p. 596.

Example 8.15: ICA Design Using the EASI Algorithm

Let us first put together a SIMULINK model that provides the desired test data for the HDL model. Let us again use a binary digital signal $s_1 = \pm 0.25\text{rect}(t/5)$ with bit length of five samples and as second signal a sine wave $s_2 = \sin(2\pi t/6)$ with period length six. These two signals are then combined with a mixing matrix (8.95) to produce the two system inputs x_1 and x_2 . For the EASI algorithm we basically have to update all four coefficients of matrices B and H . With $I = [1\ 0; 0\ 1]$ we get the following equations for H :

$$\begin{aligned} H_{1,1} &= 1 - y_1 y_1 + f_1 y_1 - y_1 f_1 = 1 - y_1 y_1 \\ H_{1,2} &= 0 - y_1 y_2 + f_1 y_2 - y_1 f_2 \\ H_{2,1} &= 0 - y_2 y_1 + f_2 y_1 - y_2 f_1 \\ H_{2,2} &= 1 - y_2 y_2 + f_2 y_2 - y_2 f_2 = 1 - y_2 y_2 \end{aligned}$$

where $f_k = f(y_k)$ is the nonlinearity applied to y_k . We get the following update equations for B :

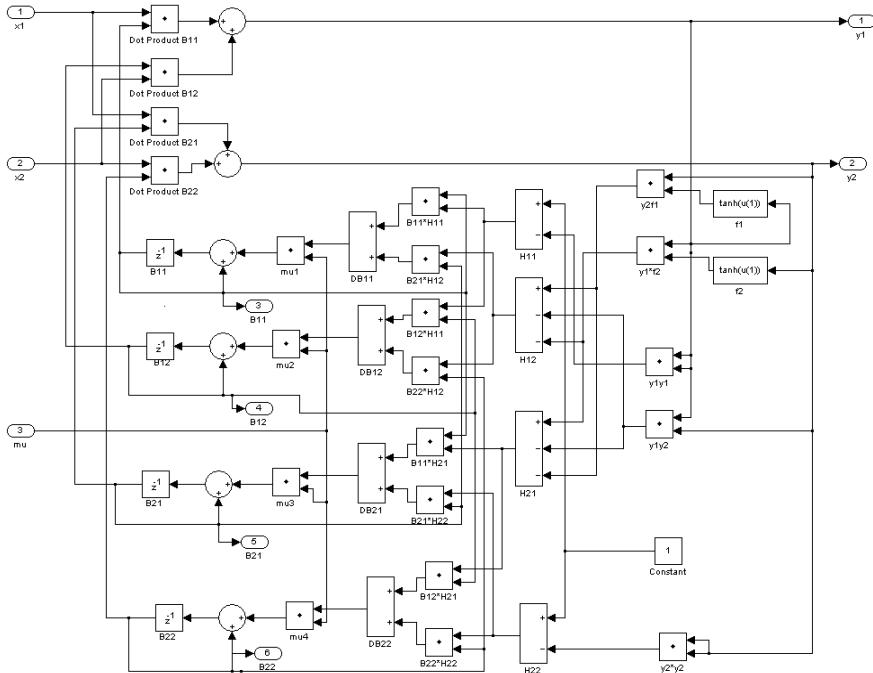


Fig. 8.42. A two channel EASI system

$$\Delta B_{1,1} = \mu (B_{1,1}H_{1,1} + H_{1,2}B_{2,1})$$

$$\Delta B_{1,2} = \mu (B_{1,2}H_{1,1} + H_{1,2}B_{2,2})$$

$$\Delta B_{2,1} = \mu (B_{1,1}H_{2,1} + H_{2,2}B_{2,1})$$

$$\Delta B_{2,2} = \mu (B_{1,2}H_{2,1} + H_{2,2}B_{2,2})$$

We now have all the equations together to build the EASI system and the overall system in SIMULINK is shown in Fig. 8.42. In the SIMULINK simulation shown in Fig. 8.43a the learning rate was 1/16 for the first 100 clock cycles and then gradually reduced to zero in the next 100 cycles, and remained at zero for the last 100 clock cycles. As expected, this ICA system is much more robust than the previously discussed PCA or Herault and Jutten ICA systems. We can switch the inputs, change the scale of the amplitudes, and still get good convergence. We can also substitute the \tanh nonlinearity by a simpler one such as the $f(y) = y \forall |y| \leq 1$ type or the signum function; see Fig. 8.41. At a cost of a little more residual noise the signals are still separated as the simulation in Fig. 8.43b shows.

The VHDL design⁶ for an ICA design using the EASI algorithm is shown in the following listing:

```

LIBRARY ieee; USE ieee.std_logic_1164.ALL;
PACKAGE n_bit_int IS -- User defined types
  SUBTYPE SLV32 IS STD_LOGIC_VECTOR(31 DOWNTO 0);

```

⁶ The equivalent Verilog code `ica.v` for this example can be found in Appendix A on page 866. Synthesis results are shown in Appendix B on page 881.

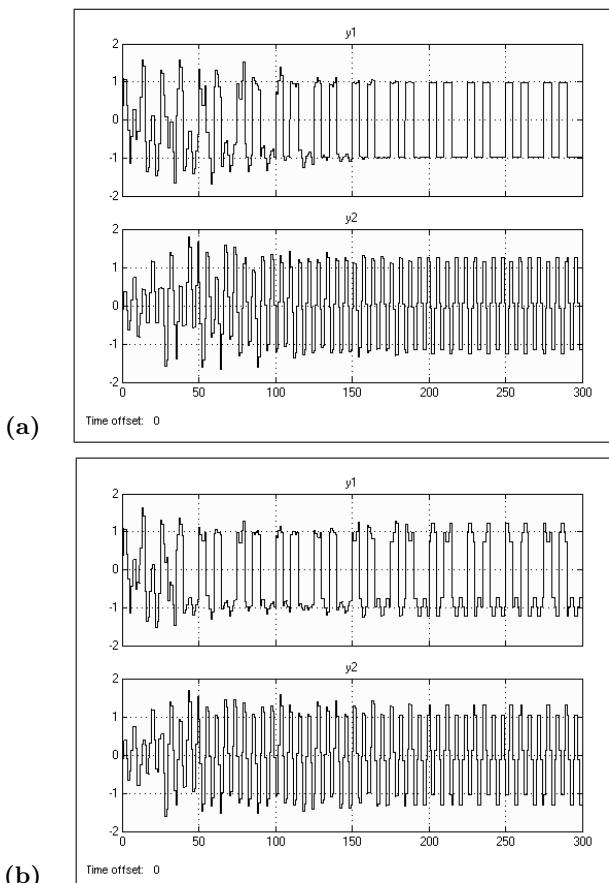


Fig. 8.43. EASI simulation in SIMULINK. (a) Using the `tanh` nonlinearity. (b) Using the `sign` nonlinearity

```

END n_bit_int;
LIBRARY work; USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL; USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

LIBRARY ieee_proposed;
USE ieee_proposed.fixed_float_types.ALL;
USE ieee_proposed.fixed_pkg.ALL;
USE ieee_proposed.float_pkg.ALL;
-----
ENTITY ica IS                                     -----> Interface
PORT (clk      : IN STD_LOGIC; -- System clock
      reset   : IN STD_LOGIC; -- System reset
      ...
      );
  
```

```

        s1_in : IN SLV32;      -- 1. signal input
        s2_in : IN SLV32;      -- 2. signal input
        mu_in : IN SLV32;      -- Learning rate
        x1_out : OUT SLV32;    -- Mixing 1. output
        x2_out : OUT SLV32;    -- Mixing 2. output
        B11_out : OUT SLV32;   -- Demixing 1,1
        B12_out : OUT SLV32;   -- Demixing 1,2
        B21_out : OUT SLV32;   -- Demixing 2,1
        B22_out : OUT SLV32;   -- Demixing 2,2
        y1_out : OUT SLV32;    -- 1. component output
        y2_out : OUT SLV32);   -- 2. component output
END;
-----
ARCHITECTURE fpga OF ica IS

CONSTANT a11 : SFIXED(15 DOWNTO -16) :=
                      TO_SFIXED(0.75, 15,-16);
CONSTANT a12 : SFIXED(15 DOWNTO -16) :=
                      TO_SFIXED(1.5, 15,-16);
CONSTANT a21 : SFIXED(15 DOWNTO -16) :=
                      TO_SFIXED(0.5, 15,-16);
CONSTANT a22 : SFIXED(15 DOWNTO -16) :=
                      TO_SFIXED(0.333333, 15,-16);
CONSTANT one : SFIXED(15 DOWNTO -16) :=
                      TO_SFIXED(1.0, 15,-16);
CONSTANT negone : SFIXED(15 DOWNTO -16) :=
                      TO_SFIXED(-1.0, 15,-16);
SIGNAL s, s1, s2, x1, x2, B11, B12, B21, B22, mu :
                      SFIXED(15 DOWNTO -16) := (OTHERS => '0');

BEGIN

s1 <= TO_SFIXED(s1_in, s); -- Redefine bits as
s2 <= TO_SFIXED(s2_in, s); -- signed FIX 16.16 number
mu <= TO_SFIXED(mu_in, s);

P1: PROCESS (reset, clk, s1, s2) -- ICA using EASI
VARIABLE f1, f2, y1, y2, H11, H12, H21, H22, DB11, DB12,
          DB21, DB22 : SFIXED(15 DOWNTO -16) := (OTHERS => '0');
BEGIN
    IF reset = '1' THEN -- Reset x register and set B=I
        x1 <= (OTHERS => '0'); x2 <= (OTHERS => '0');
        B11 <= one; B12 <= (OTHERS => '0');
        B21 <= (OTHERS => '0'); B22 <= one;
    ELSIF rising_edge(clk) THEN
        -- Mixing matrix
        x1 <= resize(a11*s1+a12*s2,s,fixed_wrap,fixed_truncate);
        x2 <= resize(a21*s1-a22*s2,s,fixed_wrap,fixed_truncate);
        -- New y values first
        y1 := resize(x1*B11+x2*B12,s,fixed_wrap,fixed_truncate);
        y2 := resize(x1*B21+x2*B22,s,fixed_wrap,fixed_truncate);
        -- Compute the H matrix
        f1 := y1; -- Build tanh approximation function for f1
        IF y1 > one THEN f1 := one; END IF;
    END IF;
END;

```

```

IF y1 < negone THEN f1 := negone; END IF;
f2 := y2; -- Build tanh approximation function for f2
IF y2 > one THEN f2 := one; END IF;
IF y2 < negone THEN f2 := negone; END IF;
H11:=resize(one - y1*y1,s,fixed_wrap,fixed_truncate);
H12:=resize(f1*y2-y1*y2-y1*f2,s,fixed_wrap,fixed_truncate);
H21:=resize(f2*y1-y2*y1-y2*f1,s,fixed_wrap,fixed_truncate);
H22:= resize(one - y2*y2,s,fixed_wrap,fixed_truncate);
-- Update matrix Delta B
DB11:=resize(B11*H11+H12*B21,s,fixed_wrap,fixed_truncate);
DB12:=resize(B12*H11+H12*B22,s,fixed_wrap,fixed_truncate);
DB21:=resize(B11*H21+H22*B21,s,fixed_wrap,fixed_truncate);
DB22:=resize(B12*H21+H22*B22,s,fixed_wrap,fixed_truncate);
-- Store update matrix B in registers
B11 <= resize(B11 + mu*DB11,s,fixed_wrap,fixed_truncate);
B12 <= resize(B12 + mu*DB12,s,fixed_wrap,fixed_truncate);
B21 <= resize(B21 + mu*DB21,s,fixed_wrap,fixed_truncate);
B22 <= resize(B22 + mu*DB22,s,fixed_wrap,fixed_truncate);
-- Register y output
y1_out <= to_slv(y1);
y2_out <= to_slv(y2);
END IF;
END PROCESS;

x1_out <= to_slv(x1); -- Redefine bits as 32 bit SLV
x2_out <= to_slv(x2);
B11_out <= to_slv(B11);
B12_out <= to_slv(B12);
B21_out <= to_slv(B21);
B22_out <= to_slv(B22);

END fpga;

```

The design uses the scheme found in Fig. 8.42. The design uses a 32-bit data format and internally a 16.16 signed fractional data type. After specifying the I/O ports the mixing matrix $a_{k,i}$ is defined as constant values. In the architecture first the input signals are redefined in **sfixed** format that should not need any resources. The EASI algorithm is implemented with a single **PROCESS** statement. First registered x_k are set to zero and the weight registers $\mathbf{B} = \mathbf{I}$ are initialized with the unity matrix. After the **rising_edge** statement the mixing is applied, followed by the y_k component including the tanh approximation through $f(y) = y \forall |y| \leq 1$ type. Then the computation of \mathbf{H} , $\Delta\mathbf{B}$, and \mathbf{B} follows. All arithmetic is done with the **fixed_wrap** and **fixed_truncate** attributes to avoid the extra hardware cost of the default threshold operation. The **PROCESS** also includes the output assignments to y_k since we like to store these in registers to measure the registered performance of the design. Finally some more internal data are assigned to output pins for monitoring.

The design uses 2275 LEs, 172 embedded multipliers, and has a registered performance of **Fmax**=17.87 MHz using the TimeQuest slow 85C model.

The SIMULINK simulation of the EASI ICA from Fig. 8.43 is now reproduced in HDL, see Fig. 8.44. Learning rate was constant for the first 100 clock cycles (0-10 μ s) and then gradually reduced to zero in the next 100 cycles (10-20 μ s), and remained at zero for the last 100 clock cycles (20-30 μ s). At this point

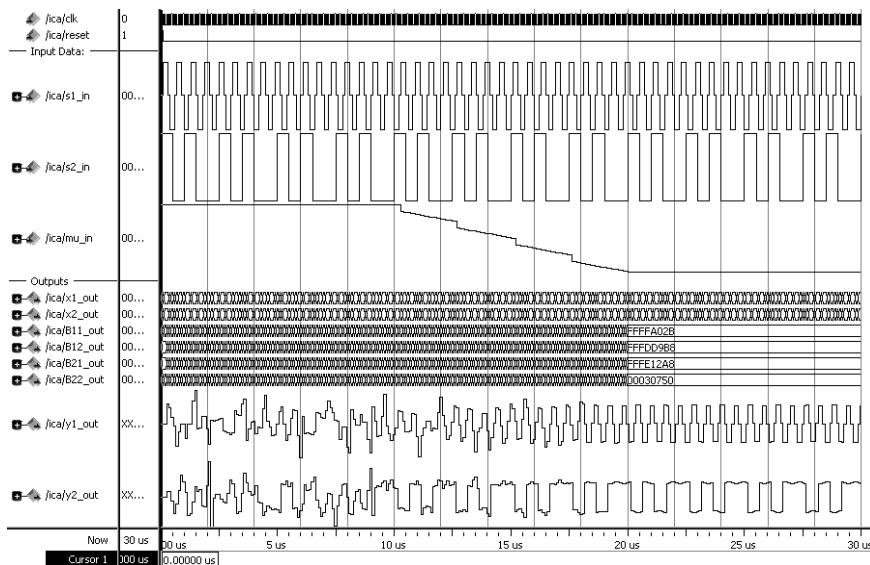


Fig. 8.44. EASI simulation in HDL. Learning (0-10 μ s), decrease learning rate (10-20 μ s), and stable output. (20-30 μ s)

in the simulation we see the signal separation into the two original signals.

8.15

If we compare the EASI algorithm with the GHA PCA design we see that we need about the same hardware resources and get the same registered performance, but the EASI algorithm is much more robust and can separate many more signals than the PCA algorithm.

8.9.4 Alternative BSS Algorithms

The EASI was generalized by Karhunen et al. [321] using two nonlinearities to introduce even more higher order statistic. These will lead to the generalized EASI algorithm that can be implemented as follows in MATLAB:

```
yk = B * x(:,k);
H = I - yk*yk' + f(yk,n1)*f(yk,n2)' - f(yk,n2)*f(yk,n1)';
B = B + mu * H * B;
```

where the nonlinearities shown in Fig. 8.41 can be selected via the parameters **n1** and **n2**.

The “trick” used in the EASI algorithm to combine learning and whitening to have a robust algorithm can also be used with other ICA algorithms. For instance the nonlinear subspace learning algorithm [313, p. 262] can be modified in such a way and we would get

```

yk = B * x(:,k);
H = f(yk,n1)*yk' - f(yk,n1)*f(yk,n1)' + I - yk*yk';
B = B + mu * H;

```

which also does not need the inputs to be white.

There are many more ICA algorithms available and, given the fact that each can be implemented with a different nonlinearity, we have many choices. For a microprocessor or MATLAB the FastICA [322] algorithm for instance is very popular since it converges fast and is robust; however, from an implementation standpoint the required symmetric orthogonalization:

```
B = B * real(inv(B' * B)^(1/2))
```

would be very expensive to build in hardware [320].

There are also a couple of non-ICA algorithms that have been successful used in the BSS problem such as SOBI or AMUSE to name just two. These algorithms only use second order statistics, no nonlinearities, but require the computation of time delayed (cross) correlation information. One algorithm that is particular easy to implement and gave good results in our simulations is the *Algorithm for Multiple Unknown Signals Extraction* (AMUSE) and it works as follows:

```

% Calculate the covariance matrix with delay
czd = corrd(z, 1);
czd=(czd+czd')/2; % make it symmetric
% Calculate the eigenvalues and eigenvectors of cov. matrix
[Ed, Dd] = eig(czd);
%apply 2. reconstruction mixing to the y data
yy=real(Dd^-0.5 * Ed'* z);

```

z has been the pre-whitening mixing signals x . `corrd` is a function that builds the correlation matrix between $z(t)$ and $z(t-\tau)$, where $\tau = 1$ worked fine in most simulations. In MATLAB we compute

```

function [C] = corrd(x, d)
[r l]=size(z); u=z(:,1:l-d); v=z(:,1+d:l);
for k=1:r
    for n=1:r
        C(k,n) = xcorr(u(k,:),v(n,:),0,'biased');
    end
end

```

We have seen that ICA methods offer a wider choice in solving tasks such as the BSS problem. However, there are still many challenges with the use of ICA algorithms. These start with the selection of the algorithm, its performance, and implementation cost. This also depends on the nonlinearity chosen that is closely related to the kurtosis of the signals; remember that the algorithms use the Hebbian learning method

$$\Delta \mathbf{w} = \sigma f(\mathbf{wx}) \quad (8.101)$$

and the sign $\sigma = \pm$ depends on whether the signals are sub- or supergaussian [323]. Remember from the kurtosis discussion that technical signals are most often subgaussian while natural signals such as speech signals are supergaussian; see Table 8.1, p. 539. The next problem comes from the fact that ICA other than PCA do not show the PC in a special order; they may appear in any order at the output and unlike the PCA the number of sources and output signals must be the same.

Comparing different algorithm is also challenging, when considering the number of learning steps, computation effort, and residual errors. Many methods are in use here ranging from visual inspection to measuring the product of mixing matrix times reconstruction matrix $\mathbf{AB} \rightarrow \mathbf{1}$. This product does not give a unit matrix since the components are in random order at the output. However, the product should only show a single nonzero value in each row and column since then a de-correlation has taken place.

8.10 Coding of Speech and Audio Signals

For many years the compression of speech signals was one of the most researched topics in DSP. Some of the most sophisticated and highly adaptive algorithms in DSP have been developed for speech and audio compression. Most algorithms require a lot of information not only from DSP but also, for instance, from psychoacoustics, that is a research topic of its own [324]. Clearly, an improvement of a few percent in speech coding multiplied by millions of users in a telephone system has a huge economical impact. Not surprisingly many coders have been proposed and several telecom standards defined ranging from high quality audio coders to text-to-speech systems. Table 8.4 gives an overview of the most often used methods and standards. Starting with linear pulse code modulation (PCM) high quality speech signals were sampled at 8 kHz with 12- to 16-bit precision resulting in a data rate of 96 – 128 Kbits per second. The first waveform compression scheme standardized by ITU-T called G.711 used a logarithmic type amplitude compression [325]. A 13- or 14-bit speech signal could be reduced to 8 bits per sample without any noticeable degradation. This G.711 scheme like many others, e.g., MP3, depends on what we call psychoacoustic effects: our auditory system has certain perception properties that the coding scheme uses [324]. Let us have a look first at the G.711 system that can be used for audio and speech compression.

8.10.1 A- and μ -Law Coding

For G.711 the psychoacoustic fact used is that amplitude changes at high sound or dB levels are less likely to be detected than changes at lower dB

Table 8.4. Comparison of audio and speech compression methods [326, 327]

Algorithm	PCM	A-law	ADPCM	LPC-10	MPEG 1 level 1,2	MPEG 1 level 3
Year	1948	1972	1984	1984	1993	1993
Standard		G.711	G.722	FS1015	MPEG	MPEG
Type	Wave coder	Wave coder	Wave coder	Vocoder	Subband	Subband + Transform
Compression ratio	1:1	1:2	1:4	1:25	1:11	1:11
Complexity	low	low	fair	high	high	high
Speech quality	high	high	good	good	good	high
Audio quality	high	good	good	low	good	high

sound levels. Or, putting it differently, at low dB the auditory system is more sensitive to quantization noise than at high amplitude levels. Exactly this is used in the G.711 standard. Small amplitudes are coded with higher resolution or less quantization noise than amplitudes at high amplitude levels. Two different proposals are summarized in G.711. The μ -law uses the nonlinearity

$$y = \frac{\ln(1 + \mu|x|)}{\ln(1 + \mu)} \text{sign}(x) \quad (8.102)$$

where typically $\mu = 255$ is used. The second option is the A-law, more popular in Europe, and it can be described by

$$y = \begin{cases} \frac{A|x|}{1+\ln(A)} \text{sign}(x) & 0 \leq |x| \leq \frac{1}{A} \\ \frac{\ln(A|x|)}{1+\ln(A)} \text{sign}(x) & \frac{1}{A} \leq |x| < 1 \end{cases} \quad (8.103)$$

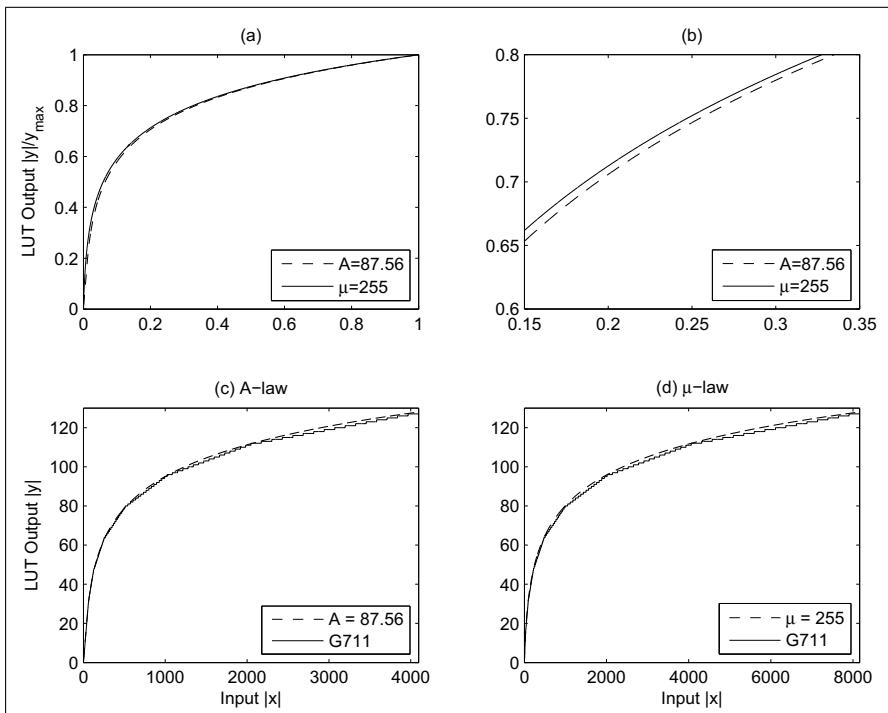
where $A = 87.56$ is used in G.711. Figure 8.45 compares the two nonlinearities. The compression curves are very similar, although the input data ranges are different. For A-law $|x| \leq 4095$, i.e., a 12-bit data range is used, while μ -law $|x| < 8158$, i.e., about a 13-bit plus sign range is used.

From the implementation standpoint the A-law has an attractive feature that it can also be seen as a short floating-point format. The output format consists of 8 bits arranged as shown in Table 8.5. First comes the sign bit, then three bits that describe the number of left-zeros in the signed magnitude format and then the four MSBs of the magnitude. The A-law uses seven segments, and the basic computation for the four magnitudes bits is

$$y_k = \lfloor x/2^k \rfloor \quad (8.104)$$

Segments 2 to 7 include 16 output values, and the first segment includes 32 values. Table 8.5 gives the full coding of all seven different left-zero exponents.

From the entropy coding standpoint we can also argue that the A- or μ -law takes better care of the available channel capacity by producing a flatter histogram than the original speech data. This can easily be verified via a short MATLAB simulation that is shown in Fig. 8.46. Little or no difference between

**Fig. 8.45.** A- and μ -law comparison**Table 8.5.** A-law coding using the short floating-point format. (s=sign bit, a,b,c,... single data bits)

Segment number	Input linear PCM	Output 8-bit A-law codes
7	s1abccedfg hij	s111abcd
6	s01abccedfg hi	s110abcd
5	s001abccedfg h	s101abcd
4	s0001abcde fgh	s100abcd
3	s00001abcde fg	s011abcd
2	s000001abcde f	s010abcd
1	s000000abcde f	s00abcd e

the original signals shown in Fig. 8.46a and the en/decoded A-law signal in Fig. 8.46b can be seen. The histogram of the original signal and the encoded signal however look quite different.

Let us now implement the G.711 standard in a small HDL design.

Example 8.16: A-law Design

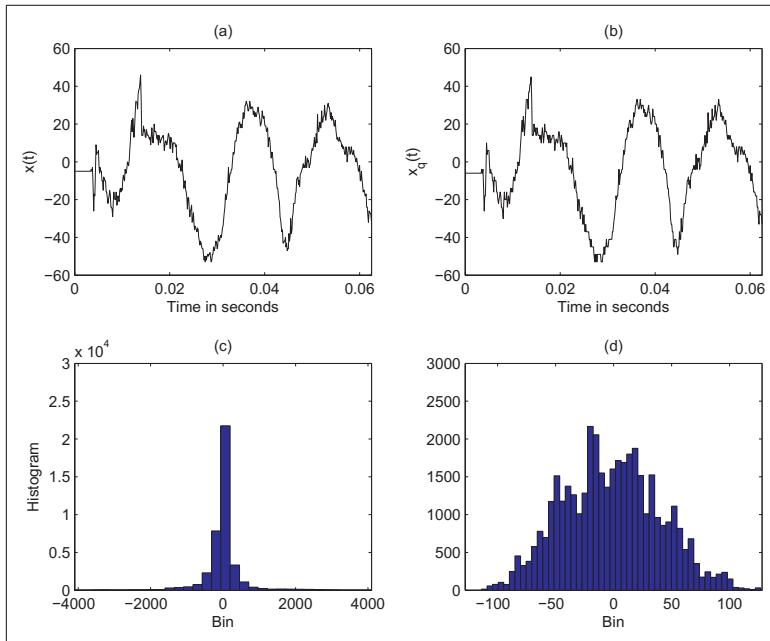


Fig. 8.46. Histogram of original speech signal and the A-law encoded signal. (a) Original speech signal first 0.06 s. (b) The A-law encoded and decoded signal. (c) Histogram of the original speech signal. (d) Histogram of the encoded signal

The VHDL design⁷ for a G.711 encoder and decoder is shown in the following listing:

```
-- G711 includes A and mu-law coding for speech signals:
-- A ~= 87.56; |x|<= 4095, i.e., 12 bit plus sign
-- mu~=255; |x|<=8160, i.e., 14 bit
LIBRARY ieee; USE ieee.std_logic_1164.ALL;
PACKAGE n_bit_int IS                         -- User defined types
    SUBTYPE SLV8 IS STD_LOGIC_VECTOR(7 DOWNTO 0);
    SUBTYPE SLV12 IS STD_LOGIC_VECTOR(11 DOWNTO 0);
    SUBTYPE SLV13 IS STD_LOGIC_VECTOR(12 DOWNTO 0);
    SUBTYPE S13 IS INTEGER RANGE -2**12 TO 2**12-1;
END n_bit_int;
LIBRARY work; USE work.n_bit_int.ALL;

LIBRARY ieee; USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-----
ENTITY g711alaw IS
    GENERIC ( WIDTH      : INTEGER := 13);      -- Bit width
    PORT (clk      : IN STD_LOGIC;   -- System clock
```

⁷ The equivalent Verilog code `g711alaw.v` for this example can be found in Appendix A on page 869. Synthesis results are shown in Appendix B on page 881.

```

        reset : IN STD_LOGIC; -- Asynchronous reset
        x_in  : IN SLV13;      -- System input
        enc   : BUFFER SLV8;    -- Encoder output
        dec   : BUFFER SLV13;    -- Decoder output
        err   : OUT S13 := 0);  -- Error of results
END g711alaw;
-----
ARCHITECTURE fpga OF g711alaw IS

SIGNAL s, s_d  : STD_LOGIC;
SIGNAL abs_x, x, x_d, x_dd : SLV13; -- Auxiliary vectors
SIGNAL temp : SLV12;

BEGIN

s <= x_in(WIDTH-1); -- sign magnitude not 2C!
abs_x <= '0' & x_in(WIDTH-2 DOWNTO 0);
err <= abs(conv_integer('0'&dec)-conv_integer('0'&x_in));

Encode: PROCESS(abs_x, s)
BEGIN
-- Mini floating-point format encoder
CASE conv_integer('0' & abs_x) IS
WHEN 0 TO 63 =>
    enc <= s & "00" & abs_x(5 DOWNTO 1); -- segment 1
WHEN 64 TO 127 =>
    enc <= s & "010" & abs_x(5 DOWNTO 2); -- segment 2
WHEN 128 TO 255 =>
    enc <= s & "011" & abs_x(6 DOWNTO 3); -- segment 3
WHEN 256 TO 511 =>
    enc <= s & "100" & abs_x(7 DOWNTO 4); -- segment 4
WHEN 512 TO 1023 =>
    enc <= s & "101" & abs_x(8 DOWNTO 5); -- segment 5
WHEN 1024 TO 2047 =>
    enc <= s & "110" & abs_x(9 DOWNTO 6); -- segment 6
WHEN 2048 TO 4095 =>
    enc <= s & "111" & abs_x(10 DOWNTO 7);-- segment 7
WHEN OTHERS      => enc <= s & "0000000"; -- + or - 0
END CASE;
END PROCESS;

Decode: PROCESS(enc, s)
BEGIN
-- Mini floating point format decoder
CASE conv_integer('0' & enc(6 DOWNTO 4)) IS
WHEN 0 | 1 =>
    dec <= s & "000000" & enc(4 DOWNTO 0) & "1";
WHEN 2      =>
    dec <= s & "000001" & enc(3 DOWNTO 0) & "10";
WHEN 3      =>
    dec <= s & "00001" & enc(3 DOWNTO 0) & "100";
WHEN 4      =>
    dec <= s & "0001" & enc(3 DOWNTO 0) & "1000";
WHEN 5      =>
    dec <= s & "001" & enc(3 DOWNTO 0) & "10000";

```

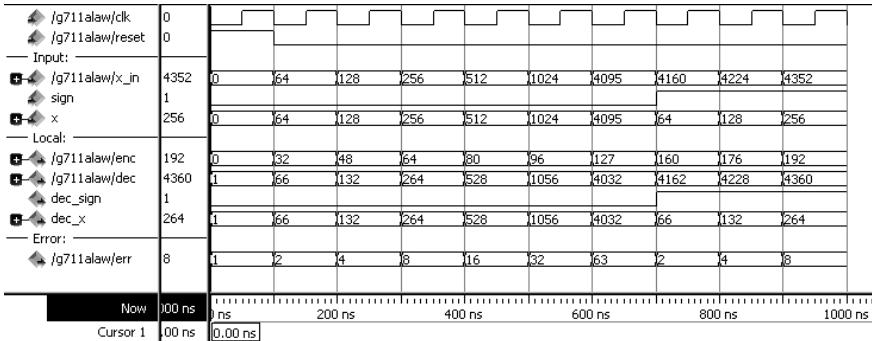


Fig. 8.47. VHDL simulation of A-law CODEC

```

WHEN 6      =>
    dec <= s & "01" & enc(3 DOWNTO 0) & "100000";
WHEN OTHERS =>
    dec <= s & "1" & enc(3 DOWNTO 0) & "1000000";
END CASE;
END PROCESS;

END fpga;

```

The design uses the encoding scheme found in Table 8.5 for encoder and decoder. Since we did not use registers no delay between input, encoded, and decoded data occurs. We notice from the simulation shown in Fig. 8.47 the small quantization error for small input values and a much larger error for large amplitude values. The design uses 70 LEs and no embedded multiplier. The registered performance cannot be measured since no registers are used.

8.16

The CD includes sound samples coded in 16-bit `SpeechPCM16bit.wav`, `Speech_A_LAW8bit.wav` and `Speech_PCM8bit.wav` for comparison. The noise level in the 8-bit PCM is perceived as much higher than the 8-bit A-law coded signals that have almost the original sound quality. Note since the WAF file format used by MATLAB has 16-bit 8 kHz (128 kbps) PCM data, the actual file sizes on CD are the same for all files.

8.10.2 Linear and Adaptive PCM Coding

The next class of coder is based on the fact that audio and speech signals usually only change gradually without sharp edges as, for instance, we find in images. As a result the correlation between neighboring samples is quite large. If we therefore code the difference between the current and previous sample the amplitude we need to code should be reduced and therefore the coding effort too. From a practical perspective it is important to use a “good” quantizer that takes advantage of the new shape of the amplitude. We may

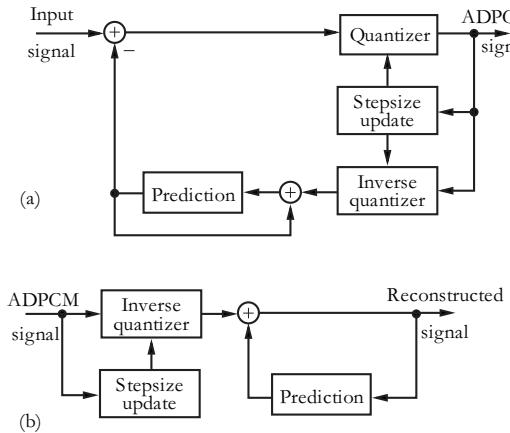


Fig. 8.48. ADPCM. (a) Encoder. (b) Decoder

implement a system that only allows very small changes. These small changes are accumulated such that we only need to transmit or store the sign of our delta operation. If we observe a larger amplitude change then the delta method may need several clock cycles to adjust. We would therefore use some oversampling rate above the Nyquist rate such as 4 or 16 that the error does not become too large. Such a scheme is usually called *Delta Modulation* (DM) in the literature.

Another idea is that we don't use oversampling, but instead use a quantizer with a few bits that can be adjusted in the quantization step size. We may also use a few of these previous samples and we would call this a prediction scheme; see Fig. 8.2, p. 535. Our quantizer steps are adjusted to the input signals. If the input x has large changes then the difference signals will be large. In a steady state, however, when the input signal does not change much, we would have a step size similar to the DM with a much smaller quantization step size. Since this is very similar to the DM but with adaptation of the quantization step size, we call such a system *Adaptive differential pulse code modulation* (ADPCM). The overall encoder and decoder is shown in Fig. 8.48.

There are several ITU standards available such as G.726, G.727, or G.722 that use ADPCM. Compared with MP3 or LPC-10 such an ADPCM is usually much easier to implement and still achieves good coding gains when compared with A-law systems. We will use the 4-bit ADPCM as proposed by IMA as an example system since this single tap predictor seemed simpler to implement than the ITU standards [328, 329].

Example 8.17: ADPCM Design

The VHDL design⁸ for encoder and decoder as specified by IMA is shown in the following listing:

```
-- This is a ADPCM demonstration for the IMA algorithm
PACKAGE n_bits_int IS          -- User defined types
    SUBTYPE U3 IS INTEGER RANGE 0 TO 7;
    SUBTYPE U4 IS INTEGER RANGE 0 TO 15;
    SUBTYPE S5 IS INTEGER RANGE -16 TO 15;
    SUBTYPE S8 IS INTEGER RANGE -128 TO 127;
    SUBTYPE U15 IS INTEGER RANGE 0 TO 2**15-1;
    SUBTYPE S16 IS INTEGER RANGE -2**15 TO 2**15-1;
    SUBTYPE S17 IS INTEGER RANGE -2**16 TO 2**16-1;
    TYPE A0_7S5 IS ARRAY (0 TO 7) of S5;
    TYPE A0_88U15 IS ARRAY (0 TO 88) of U15;
END n_bits_int;

LIBRARY work; USE work.n_bits_int.ALL;

LIBRARY ieee; USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-----
ENTITY adpcm IS           -----> Interface
    PORT ( clk      : IN STD_LOGIC; -- System clock
           reset    : IN STD_LOGIC; -- Asynchronous reset
           x_in     : IN S16; -- Input to encoder
           y_out    : OUT U4; -- 4 bit ADPCM coding word
           p_out    : OUT S16; -- Predictor/decoder output
           p_underflow, p_overflow : OUT STD_LOGIC;
                               -- Predictor flags
           i_out    : OUT S8; -- Index to table
           i_underflow, i_overflow : OUT STD_LOGIC;
                               -- Index flags
           err      : OUT S16; -- Error of system
           sz_out   : OUT U15; -- Step size
           s_out    : OUT STD_LOGIC); -- Sign bit
END adpcm;
-----
ARCHITECTURE fpga OF adpcm IS

    -- ADPCM step variation table
    CONSTANT indexTable : A0_7S5 :=(
        -1, -1, -1, -1, 2, 4, 6, 8);

    -- Quantization lookup table has 89 entries
    CONSTANT stepsizeTable : A0_88U15 :=(
        7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 19, 21, 23, 25,
        28, 31, 34, 37, 41, 45, 50, 55, 60, 66, 73, 80, 88,
        97, 107, 118, 130, 143, 157, 173, 190, 209, 230, 253,
        279, 307, 337, 371, 408, 449, 494, 544, 598, 658, 724,
        796, 876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878,
```

⁸ The equivalent Verilog code adpcm.v for this example can be found in Appendix A on page 870. Synthesis results are shown in Appendix B on page 881.

```

2066, 2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428,
4871, 5358, 5894, 6484, 7132, 7845, 8630, 9493, 10442,
11487, 12635, 13899, 15289, 16818, 18500, 20350, 22385,
24623, 27086, 29794, 32767);

SIGNAL va, va_d : S16 := 0;-- Current signed adpcm input
SIGNAL sign : STD_LOGIC; -- Current adpcm sign bit
SIGNAL sdelta : U4; -- Current signed adpcm output
SIGNAL step : U15 := 7; -- Stepsize
SIGNAL sstep : S16; -- Stepsize including sign
SIGNAL valpred : S16 := 0; -- Predicted output value
SIGNAL index : S8 := 0; -- Current step change index

BEGIN

Encode: PROCESS(clk, reset, x_in, va, sign, sdelta, step,
                valpred, index)
VARIABLE diff : S17; -- Difference val - valpred
VARIABLE p : S17 := 0; -- Next valpred
VARIABLE i : S8; -- Next index
VARIABLE delta : U3; -- Current absolute adpcm output
VARIABLE tStep : U15;
VARIABLE vpdiff : S16; -- Current change to valpred
BEGIN
    IF reset = '1' THEN -- Asynchronous clear
        va <= 0; va_d <= 0;
    ELSIF rising_edge(clk) THEN -- Store in register
        va <= x_in;
        va_d <= va; -- Delay signal for error comparison
    END IF;
    ----- State 1: Compute difference from predicted sample
    diff := va - valpred;
    sign <= '0';
    IF diff < 0 THEN
        sign <= '1'; -- Set sign bit if negative
        diff := -diff; -- Use absolute value for quantization
    END IF;
    -- State 2: Quantize by devision and
    -- State 3: compute inverse quantization
    -- Compute: delta=floor(diff(k)*4./step(k)); and
    -- vpdiff(k)=floor((delta(k)+.5).*step(k)/4);
    delta := 0; tStep := step; vpdiff := tStep/8;
    IF diff >= tStep THEN
        delta := 4; diff := diff-tStep;
        vpdiff := vpdiff + tStep;
    END IF;
    tStep := tStep/2;
    IF diff >= tStep THEN
        delta := delta + 2 ; diff := diff - tStep;
        vpdiff := vpdiff + tStep;
    END IF;
    tStep := tStep/2;
    IF diff >= tStep THEN

```

```

        delta := delta + 1; diff := diff - tStep;
        vpdiff := vpdiff + tStep;
    END IF;
-- State 4: Adjust predicted sample based on inverse
    IF sign = '1' THEN                                -- quantized
        p := valpred - vpdiff;
    ELSE
        p := valpred + vpdiff;
    END IF;
----- State 5: Threshold to maximum and minimum -----
    p_overflow <= '0'; p_underflow <= '0';
    IF p > 32767 THEN -- Check for 16 bit range
        p := 32767; -- 2^15-1 two's complement
        p_overflow <= '1';
    END IF;
    IF p < -32768 THEN
        p := -32768; -- -2^15
        p_underflow <= '1';
    END IF;
    IF reset = '1' THEN -- Asynchronous clear
        valpred <= 0;
    ELSIF rising_edge(clk) THEN
        valpred <= p;           -- Store predicted in register
    END IF;
--- State 6: Update the stepsize and index for stepsize LUT
    i_underflow <= '0'; i_overflow <= '0';
    i := index + indexTable(delta);
    IF i < 0 THEN -- Check index range [0...88]
        i := 0;
        i_underflow <= '1';
    END IF;
    IF i > 88 THEN
        i := 88;
        i_overflow <= '1';
    END IF;
    IF reset = '1' THEN -- Asynchronous clear
        step <= 0; index <= 0;
    ELSIF rising_edge(clk) THEN
        step <= stepsizeTable(i);
        index <= i;
    END IF;
    IF sign = '1' THEN
        sdelta <= delta + 8;
    ELSE
        sdelta <= delta;
    END IF;
END PROCESS;

y_out  <= sdelta;      -- Monitor some test signals
p_out  <= valpred;
i_out  <= index;
sz_out <= step;
s_out  <= sign;

```

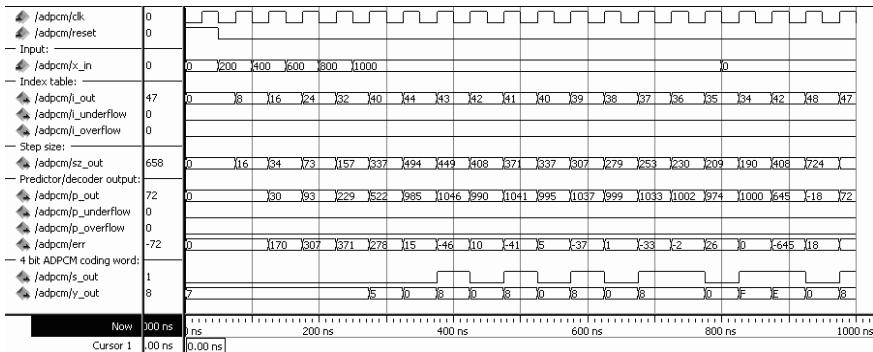


Fig. 8.49. VHDL simulation of ADPCM CODEC

```
err <= va_d-valpred;
```

```
END fpga;
```

The design uses the encoding scheme found in Fig. 8.48 for encoder and decoder. The circuit HDL description starts with the I/O ports followed by the look-up tables for step variations and step size. As expected the quantization table increases first in small steps (i.e., 1) and for larger values with larger steps (ca. 3K). Only the encoder is shown, since the decoder can be seen as the last steps of the encoder; see Fig. 8.48. First the input and a one clock cycle delayed version are stored in registers. The encoder computes in the first step the difference from the new value to the predicted value. The quantization is done by division and the inverse quantization are calculated. Next in step 4 the predicted sample is updated using the inverse quantized value. In step 5 the predicted values run through a thresholding procedure to avoid overflow. In the final step 6 we update the step size used and store the step size and the index used in registers. The encoder output is concatenated with the sign to form the output word **sdelta**. Finally we assign the signals that are shown in the simulation to the output pins.

In the simulation shown in Fig. 8.49 the input signal is a ramp followed by a constant values 1000. We see that the step size **sz_out** is first increased and then gradually reduced when the input is constant. The error is high in the ramp phase and much smaller in the phase when the input is constant.

The design uses 531 LEs, no embedded multipliers, and has a registered performance of **Fmax=49.5 MHz** using the **TimeQuest slow 85C** model. 8.17

The CD includes sound samples:

- SpeechPCM16bit.wav, i.e., 16-bit coded original speech signal
- Speech_PCM4bit.wav, 4-bit linear PCM
- Speech_APCM4.wav, 4-bit ADPCM coded
- Speech_PCM4Lloyd.wav, 4-bit Lloyd coded

All files are in the **WAV** file format. The Lloyd method [330] computes the codebook that minimizes the quantization noise based on the number of bits

used and training data provided; see Fig. 8.50. In MATLAB this is computed via

```
[partition, codebook, distor, rel_distor] = ...
lloyds(training_set, ini_codebook, [], 3);
%% encode + decode
[indx, quant, distor] = quantiz(sig, partition, codebook);
```

The noise level in the 4-bit linear PCM is too high. Better is the 4-bit Lloyd but still higher than the 4-bit ADPCM signal that has almost the original sound quality. Note that since the WAF file format used by MATLAB has 16-bit 8 kHz (128 kbps) PCM data, the actual file sizes on CD are the same for all files.

8.10.3 Coding by Modeling: The LPC-10e Method

The method discussed so far can be used for speech and audio compression. If we like to achieve a higher compression for speech signals than 4:1 we need to give up the audio coding and use a coder that tries to rebuild the vocal tract and just codes its parameters, i.e., pitch, formants, short time spectra, and vocal tract transfer functions. Speech production can usually be split roughly into voiced (the vowels) and unvoiced speech (no vowels), where the vocal tract is modeled by its filter shape and the exiting is an impulse train (pitch) or a random noise; see Fig. 8.51. Such a coding system would be called a *linear predictive coder* and LPC-10e in particular is one of the most popular and powerful speech coder available. In general such systems are called *vocoders* and the US DOD standard FS1015 at 2400 bps produces very good results; even an 800 bps vocoder has been successfully tested [331]. Below that, only text-to-speech systems may achieve higher compression ratios. The drawback of the LPC vocoder is that, for general audio compression (e.g., music), the results are not satisfactory.

The popular LPC-10e coding scheme for a 2400 bps transmission would encode the signals as follows. The input signals would be sampled at 8 kHz and split into nonoverlapping blocks of 180 samples, or 44.44 frames per second. At 2400 bps we will only have 54 bits to code all parameters of our speech model. It would use a vocal tract filter model with ten filter coefficients from a lattice type FIR filter. The pitch would be computed by an average magnitude difference function that is an estimate for the autocorrelation by computing

$$y(k) = \sum_m |x(n+m) - x(n+m-k)| \quad (8.105)$$

If x is periodic with N then the function $y(k)$ will show notches at $k = N, 2N, \dots$ which is used as the pitch period of the speech signal. We would code this pitch with seven bits in LPC-10e. For the filter coefficients we would use more bits for lower partial correlation (PARCOR) coefficients and less

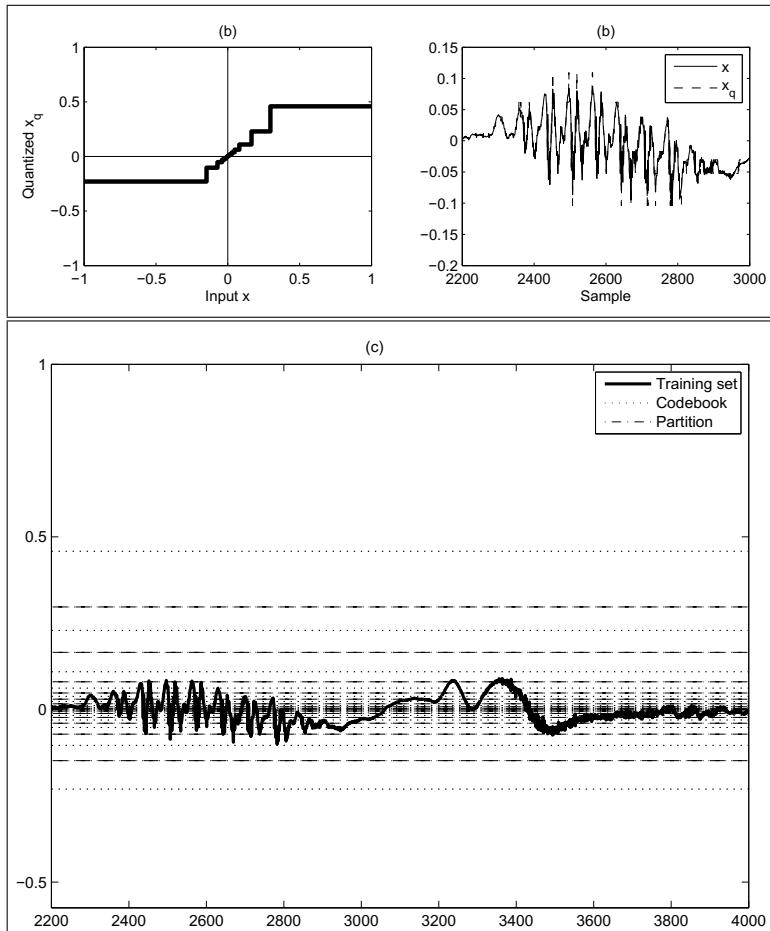


Fig. 8.50. Lloyd optimized quantizer. (a) Quantization LUT a.k.a. codebook and partition. (b) Comparison of input and quantized wave form. (c) Optimized quantization code partition and codebook

bits for the last lattice coefficients (total $5 \times 4 + 4 \times 4 + 3 + 2 = 41$ bits) and an extra 5 bits for the filter gain factor. One bit is left for synchronization and the total becomes $41 + 7 + 5 + 1 = 54$ bits per frame. The LPC-10e seemed to be a high complexity algorithm and a 7.8 kbps and 2.4 kbps PDSP implementation can be found, for instance, in ADSP Application Handbook Vol. 2 [332].

8.10.4 MPEG Audio Coding Methods

The audio coders in the MPEG standards have been quite successful in recent years. In fact the level three coder of the MPEG 1 coder usually called MP3 is

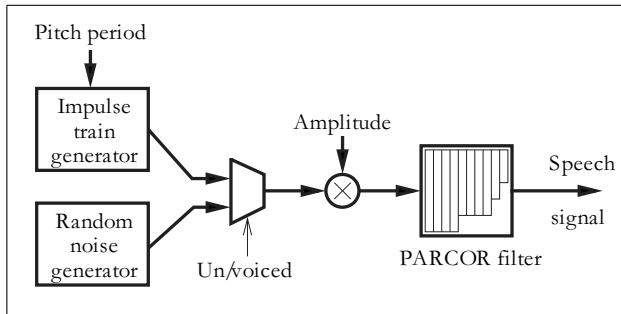


Fig. 8.51. Simplified speech production model

the most often used audio format on the Internet today [333]. Since MPEG is intended to produce high quality audio signals and not just compressed speech signals the achieved compression ratios are not as high as with the LPC-10e vocoder. Another factor in the complexity is the sampling rate of audio that is usually 44.1 kHz for CD quality and just 8 kHz for speech signals. Nevertheless the MP3 produces excellent quality at 11:1 compression ratio. The main processing done in the MPEG coder is a cosine modulated 32 channel uniformed spaced filter bank of length 513. The impulse response of the bandpass filters are

$$h_k[n] = h[n] \cos \left((2k+1) \left(\frac{n}{M} - \frac{1}{2} \right) \frac{\pi}{2} \right) \quad \begin{cases} n = 0, 1, \dots, 512 \\ k = 0, 1, \dots, 31 \end{cases} \quad (8.106)$$

where the prototype filter has a -90 dB stopband suppression. This equally spaced filter bank will have a channel band width of around 700 Hz. This frequency resolution is improved by the level 3 encoder that uses a modified DCT in each channel for 12 to 36 samples improving the frequency resolution to ca. 40 Hz. In addition the level 3 adds a non-uniform Huffman coding as the final compression stage.

Since all audio signals need to be compressed in MPEG we can rely only on basic psychoacoustic effects, but no vocoder system can be used. The most important psychoacoustic effect that has been used in MPEG coders is frequency masking. Here a sine wave with a large amplitude will mask other frequencies in the neighborhood. Our auditory system is not capable of deciding whether the neighboring frequency signal are indeed there or not. Since the auditory system does not hear these mask signals we do not need to include these in the coding. Typically a 12 dB per octave roll-off to higher frequencies is assumed and a 24 dB per octave roll-off to lower frequencies [334]. This masking is determined via 512 (MPEG 1+2) or 1024 (MPEG 3) length FFTs and used for the quantization of the coefficients. Figure 8.52 gives an overview of the MP3 encoder.

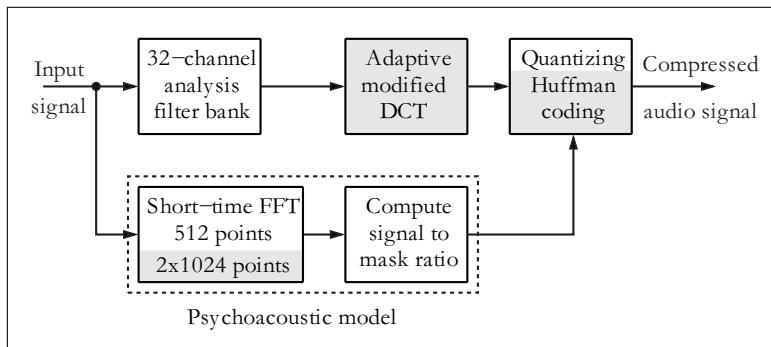


Fig. 8.52. The MPEG 1 level three a.k.a. MP3 encoder. Gray indicates improvements compared to level one

The MP3 coder design with 32 length-513 filters, DCT, and Huffman coder will make this a challenging design. In fact MP3 was at first considered too complex to be built in realtime and level 1 was consider more practical.

Exercises

Note: If you have no prior experience with the Quartus II software, refer to the case study found in Sect. 1.4.3, p. 32. If not otherwise noted use the EP4CE115F29C7 from the Cyclone IV E family for the Quartus II synthesis evaluations.

8.1: Suppose the following signal is given:

$$x[n] = A \cos[2\pi n/T + \phi].$$

- (a) Determine the power or variance σ^2 .
- (b) Determine the autocorrelation function $r_{xx}[\tau]$.
- (c) What is the period of $r_{xx}[\tau]$?

8.2: Suppose the following signal is given:

$$x[n] = A \sin[2\pi n/T + \phi] + n[n],$$

where $n[n]$ is a white Gaussian noise with variance σ_n^2 .

- (a) Determine the power or variance σ^2 of the signal $x[n]$.
- (b) Determine the autocorrelation function $r_{xx}[\tau]$.
- (c) What is the period of $r_{xx}[\tau]$?

8.3: Suppose the following two signals are given:

$$x[n] = \cos[2\pi n/T_0] \quad y[n] = \cos[2\pi n/T_1].$$

- (a) Determine the cross-correlation function $r_{xy}[\tau]$.
- (b) What is the condition for T_0 and T_1 that $r_{xy}[\tau] = 0$?

8.4: Suppose the following signal statistics have been determined:

$$\mathbf{R}_{xx} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \quad \mathbf{r}_{dx} = \begin{bmatrix} 4 \\ 5 \end{bmatrix} \quad \mathbf{R}_{dd}[0] = 20.$$

Compute

- (a) Compute \mathbf{R}_{xx}^{-1} .
- (b) The optimal Wiener filter weight.
- (c) The error for the optimal filter weight.
- (d) The eigenvalues and the eigenvalue ratio.

8.5: Suppose the following signal statistics for a second order system are given:

$$\mathbf{R}_{xx} = \begin{bmatrix} r_0 & r_1 \\ r_1 & r_0 \end{bmatrix} \quad \mathbf{r}_{dx} = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \quad \mathbf{R}_{dd}[0] = \sigma_d^2.$$

The optimal filter with coefficient should be f_0 and f_1 .

- (a) Compute \mathbf{R}_{xx}^{-1} .
- (b) Determine the optimal filter weight error as a function of f_0 and f_1 .
- (c) Determine f_0 and f_1 as a function of r and c .
- (d) Assume now that $r_1 = 0$. What are the optimal filter coefficients f_0 and f_1 ?

8.6: Suppose the desired signal is given as:

$$d[n] = \cos[2\pi n/T_0].$$

The reference signal $x[n]$ that is applied to the adaptive filter input is given as

$$x[n] = \sin[2\pi n/T_0] + 0.5 \cos[2\pi n/T_1],$$

where $T_0 = 5$ and $T_1 = 3$. Compute for a second order system:

- (a) \mathbf{R}_{xx} , \mathbf{r}_{dx} , and $\mathbf{R}_{dd}[0]$.
- (b) The optimal Wiener filter weight.
- (c) The error for the optimal filter weight.
- (d) The eigenvalues and the eigenvalue ratio.
- (e) Repeat (a)–(d) for a third order system.

8.7: Suppose the desired signal is given as:

$$d[n] = \cos[2\pi n/T_0] + n[n],$$

where $n[n]$ is a white Gaussian noise with variance 1. The reference signal $x[n]$ that is applied to the adaptive filter input is given as

$$x[n] = \sin[2\pi n/T_0],$$

where $T_0 = 5$. Compute for a second order system:

- (a) \mathbf{R}_{xx} , \mathbf{r}_{dx} , and $\mathbf{R}_{dd}[0]$.
- (b) The optimal Wiener filter weight.
- (c) The error for the optimal filter weight.
- (d) The eigenvalues and the eigenvalue ratio.
- (e) Repeat (a)–(d) for a third order system.

8.8: Suppose the desired signal is given as:

$$d[n] = \cos[4\pi n/T_0]$$

The reference signal $x[n]$, which is applied to the adaptive filter input, is given as

$$x[n] = \sin[2\pi n/T_0] - \cos[4\pi n/T_0],$$

where $T_0 = 5$. Compute for a second order system:

- (a) \mathbf{R}_{xx} , \mathbf{r}_{dx} . and $\mathbf{R}_{dd}[0]$.
- (b) The optimal Wiener filter weight.
- (c) The error for the optimal filter weight.
- (d) The eigenvalues and the eigenvalue ratio.
- (e) Repeat (a)–(d) for a third order system.

8.9: Using the 4 FIR filters given in Table 8.2 (p. 552) use C or MATLAB to compute the autocorrelation function and the eigenvalue ratio using the autocorrelation for of a (filtered) sequence of 10 000 white noise samples. For the following system length (i.e., size of autocorrelation matrix):

- (a) $L = 2$.
- (b) $L = 4$.
- (c) $L = 8$.
- (d) $L = 16$.

Hint: The MATLAB functions: `randn`, `filter`, `xcorr`, `toeplitz`, `eig` are helpful.

8.10: Using an IIR filter with one pole $0 < \rho < 1$ use C or MATLAB to compute the autocorrelation function and plot the eigenvalue ratio using the autocorrelation for a (filtered) sequence of 10 000 white noise samples. For the following system length (i.e., size of autocorrelation matrix):

- (a) $L = 2$.
- (b) $L = 4$.
- (c) $L = 8$.
- (d) $L = 16$.

(e) Compare the results from (a) to (d) with the theoretical value $\text{EVR} = (1 + \rho)/(1 - \rho)^2$ of Markov-1 processes [285].

Hint: The MATLAB functions: `randn`, `filter`, `xcorr`, `toeplitz`, `eig` are helpful.

8.11: Using the FIR filter for $\text{EVR} = 1000$ given in Table 8.2 (p. 552) use C or MATLAB to compute the eigenvectors of the autocorrelation for $L = 16$. Compare the eigenvectors with the DCT basis vectors.

8.12: Using the FIR filter for $\text{EVR} = 1000$ given in Table 8.2 (p. 552) use C or MATLAB to compute the eigenvalue ratios of the transformed power normalized autocorrelation matrices from (8.48) on page 559 for $L = 16$ using the following transforms:

- (a) Identity transform (i.e., no transform).
- (b) DCT.
- (c) Hadamard.
- (d) Haar.
- (e) Karhunen–Loéve.
- (f) Build a ranking of the transform from (a)–(e).

8.13: Using the one pole IIR filter from Exercise 8.10 use C or MATLAB to compute for 10 values of ρ in the range 0.5 to 0.95 the eigenvalue ratios of the transformed power normalized autocorrelation matrices from (8.48) on page 559 for $L = 16$ using the following transforms:

- (a) Identity transform (i.e., no transform).
- (b) DCT.
- (c) Hadamard.
- (d) Haar.
- (e) Karhunen–Loéve.
- (f) Build a ranking of the transform from (a)–(e).

8.14: Use C or MATLAB to rebuild the power estimation shown for the nonstationary signal shown in Fig. 8.15 (p. 555). For the power estimation use

- (a) Equation (8.41) page 554.
- (b) Equation (8.44) page 554 with $\beta = 0.5$.
- (c) Equation (8.44) page 554 with $\beta = 0.9$.

8.15: Use C or MATLAB to rebuild the simulation shown in Example 8.1 (p. 542). Use $T = (L - 1)240$ samples per second (i.e., $(L - 1)4$ samples per period for $h[n]$ and $x[n]$) and the following filter length:

- (a) L=2.
- (b) L=3.
- (c) L=4.
- (d) Compute the exact Wiener solution for L=3.
- (e) Compute the exact Wiener solution for L=4.

8.16: Use C or MATLAB to rebuild the simulation shown in Example 8.3 (p. 549). Use $T = (L - 1)240$ samples per second (i.e., $(L - 1)4$ samples per period for $h[n]$ and $x[n]$) and the following filter length:

- (a) L=2.
- (b) L=3.
- (c) L=4.

8.17: Use C or MATLAB to rebuild the simulation shown in Example 8.6 (p. 568) for the following pipeline configuration:

- (a) DLMS with 1 pipeline stage.
- (b) DLMS with 2 pipeline stages.
- (c) DLMS with 3 pipeline stages.
- (d) DLMS with 4 pipeline stages.

8.18: (a) Change the filter length of the adaptive filter in Example 8.5 (p. 561) to three and use 8 samples per period for $h[n]$ and $x[n]$.

- (b) Make a compilation (with the Quartus II compiler) of the HDL code for the filter.
- (c) Perform a functional simulation of the filter with the inputs $d[n]$ and $x[n]$.
- (d) Compare the results with the simulation in Exercise 8.15b and d.
- (e) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) of your $L = 3$ design using the device EP4CE115F29C7 from the Cyclone IV E family.
- (f) Repeat (e) for the EP2C35F672C6 from the Cyclone II family.

8.19: (a) Change the filter length of the adaptive filter in Example 8.5 (p. 561) to four and use 12 samples per period for $h[n]$ and $x[n]$.

- (b) Make a compilation (with the Quartus II compiler) of the HDL code for the filter.
- (c) Perform a functional simulation of the filter with the inputs $d[n]$ and $x[n]$.
- (d) Compare the results with the simulation in Exercise 8.15c and e.
- (e) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) of your $L = 4$ design using the device EP4CE115F29C7 from the Cyclone IV E family.
- (f) Repeat (e) for the EP2C35F672C6 from the Cyclone II family.

8.20: (a) Change the DLMS filter design from Example 8.6 (p. 568) pipeline of $y[n]$ only, i.e., DLMS with 1 pipeline stage. Use embedded multipliers for $p(i)$ and $xemu(i)$.

- (b) Make a compilation (with the Quartus II compiler) of the HDL code for the

filter.

- (c) Perform a functional simulation of the filter with the inputs $d[n]$ and $x[n]$.
- (d) Compare the results with the simulation in Exercise 8.17a.
- (e) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) of your D=1 design using the device EP4CE115F29C7 from the Cyclone IV E family.
- (f) Repeat (e) for the EP2C35F672C6 from the Cyclone II family.

8.21: (a) Change the DLMS filter design from Example 8.6 (p. 568) pipeline of $e[n]$ and $y[n]$ update only, i.e., DLMS with two pipeline stages. Use embedded multipliers for $p(i)$ and $xemu(i)$.

- (b) Make a compilation (with the Quartus II compiler) of the HDL code for the filter.
- (c) Perform a functional simulation of the filter with the inputs $d[n]$ and $x[n]$.
- (d) Compare the results with the simulation in Exercise 8.17b.
- (e) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) of your D=2 design using the device EP4CE115F29C7 from the Cyclone IV E family.
- (f) Repeat (e) for the EP2C35F672C6 from the Cyclone II family.

8.22: (a) Change the DLMS filter design from Example 8.6 (p. 568) to three pipeline stages. Use pipeline registers after $y[n]$ and after the embedded multipliers for $p(i)$ and $xemu(i)$. Do not use a pipeline register for $e[n]$ for this design.

- (b) Make a compilation (with the Quartus II compiler) of the HDL code for the filter.
- (c) Perform a functional simulation the filter with the inputs $d[n]$ and $x[n]$.
- (d) Compare the results with the simulation in Exercise 8.17c.
- (e) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) of your D=3 design using the device EP4CE115F29C7 from the Cyclone IV E family.
- (f) Repeat (e) for the EP2C35F672C6 from the Cyclone II family.

9. Microprocessor Design

Introduction

When you think of current microprocessors (μ Ps), the Intel Itanium processor with 592 million transistors may come to mind. Designing this kind of microprocessor with an FPGA, you may ask? Now the author has become overconfident with the capabilities of today's FPGAs. Clearly today's FPGAs will not be able to implement such a top-of-the-range μ P with a single FPGA. But there are many applications where a less-powerful μ P can be quite helpful. Remember a μ P trades performance of the hardwired solution with gate efficiency. In software, or when designed as an FSM, an algorithm like an FFT may run slower, but usually needs much less resources. So the μ P we build with FPGAs are more of the microcontroller type than a fully featured modern Intel Pentium or TI VLIW PDSP. A typical application we discuss later would be the implementation of a controller for a DWT. Now you may argue that this can be done with an FSM. And yes that is true and we basically consider our FPGA μ P design nothing else than an FSM augmented by a program memory that includes the operation the FSM will perform; see Fig. 9.1. In fact the early versions of the Xilinx PicoBlaze processor were called Ken Chapman programmable state machine (KCPSM) [335]. A complete μ P design usually involves several steps, such as the architecture exploration phase, the instruction set design, and the development tools. We will discuss these steps in the following in more details. You are encouraged to study in addition a computer architecture book; there are many available today as this is a standard topic in most undergraduate computer engineering curricula [336–342]. But before we go into details of μ P design let us first have a look back at the begin of the μ P era.

9.1 History of Microprocessors

Usually microprocessor are classified into three major classes: the general-purpose or CISC processor, reduced instruction set processors (RISC), and programmable digital signal processors (PDSP). Let us now have a brief look how these classes of microprocessor have developed.

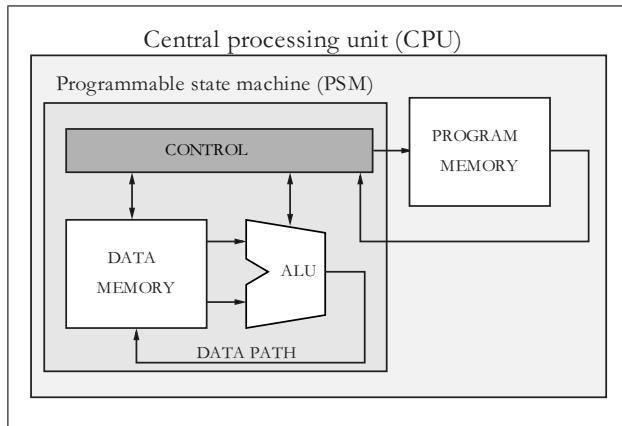


Fig. 9.1. The Xilinx KCPSM a.k.a. PicoBlaze

9.1.1 Brief History of General-Purpose Microprocessors

By 1968 the typical general-purpose minicomputers in use were 16-bit architectures using about 200 MSI chips on a single circuit board. The MSI had about 100 transistors each per chip. A popular question [343] then was: can we also build a single CPU with (only) 150, 80, or 25 chips?

At about the same time Robert Noyce and Gordon Moore, formerly with Fairchild Corp., started a new company first called NM Electronics and later renamed Intel, whose main product was memory chips. In 1969 Busicom, a Japanese calculator manufacturer, asked Intel to design a set of chips for their new family of programmable calculators. Intel did not have the manpower to build the 12 different custom chips requested by Busicom, since good IC designers at that time were hard to find. Instead Intel's engineer Ted Hoff suggested building a more-general four-chip set that would access its instructions from a memory chip. A programmable state machine (PSM) with memory was born, which is what we today call a microprocessor. After nine months with the help of F. Faggin the group of Hoffs delivered the Intel 4004, a 4-bit CPU that could be used for the BCD arithmetic used in the Busicom calculators. The 4004 used 12-bit program addresses and 8-bit instructions and it took five clock cycles for the execution of one instruction. A minimum working system (with I/O chips) could be built out of two chips only: the 4004 CPU and a program ROM. The 1 MHz clock allowed one to add multidigit BCD numbers at a rate of 80 ns per digit [344].

Hoff's vision was now to extend the use of the 4004 beyond calculators to digital scales, taxi meters, gas pumps, elevator control, medical instruments, vending machines, etc. Therefore he convinced the Intel management to obtain the rights from Busicom to sell the chips to others too. By May 1971 Intel gave a price concession to Busicom and obtained in exchange the rights to sell the 4004 chips for applications other than calculators.

Table 9.1. The family of Intel microprocessors. [345] (IA = instruction set architecture)

Name	Year intro.	MHz	IA	Process technology	#Transistors
4004	1971	0.108	4	10μ	2300
8008	1972	0.2	8	10μ	3500
8080	1974	2	8	6μ	4500
8086	1978	5-10	16	3μ	29K
80286	1982	6-12.5	16	1.5μ	134K
80386	1985	16-33	32	1μ	275K
80486	1989	25-50	32	0.8μ	1.2M
Pentium	1993	60-66	32	0.8μ	3.1M
Pentium II	1997	200-300	32	0.25μ	7.5M
Pentium 3	1999	650-1400	32	0.25μ	9.5M
Pentium 4	2000	1300-3800	32	0.18μ	42M
Xeon	2003	1400-3600	64	0.09μ	178M
Itanium 2	2004	1000-1600	64	0.13μ	592M

One of the concerns was that the performance of the 4004 could not compete with state-of-the-art minicomputers at the time. But another invention, the EPROM, also from Intel by Dov Frohamn-Bentchkovsky helped to market a 4004 development system. Now programs did not need an IC factory to generate the ROM, with the associated long delays in the development. EPROMs could be programmed and reprogrammed by the developer many times if necessary.

Some of Intel's customer asked for a more-powerful CPU, and an 8-bit CPU was designed that could handle the 4-bit BCD arithmetic of the 4004. Intel decided to build a 8008 and it also supported standard RAM and ROM devices, custom memory was no longer required as for the 4004 design. Some of the shortcoming of the 8008 design were fixed by the 8080 design in 1974 that used now about 4500 transistors. In 1978 the first 16-bit μ P was introduced, the 8086. In 1982 the 80286 followed: a 16-bit μ P but with about six times the performance of the 8086. The 80386, introduced in 1985, was the first μ P to support multitasking. In the 80387 a mathematics coprocessor was added to speed up floating-point operations. Then in 1989 the 80486 was introduced with an instruction cache and instruction pipelining as well as a mathematics coprocessor for floating-point operations. In 1993 the Pentium family was introduced, with now two pipelines for execution, i.e., a superscalar architecture. The next generation of Pentium II introduced in 1997 added multimedia extension (MMX) instructions that could perform some parallel vector-like MAC operations of up to four operands. The Pentium 3 and 4 followed with even more advanced features, like hyperthreading and SSE instructions to speed up audio and video processing. The largest processor as of 2006 is the Intel Itanium with two processor cores and a whopping

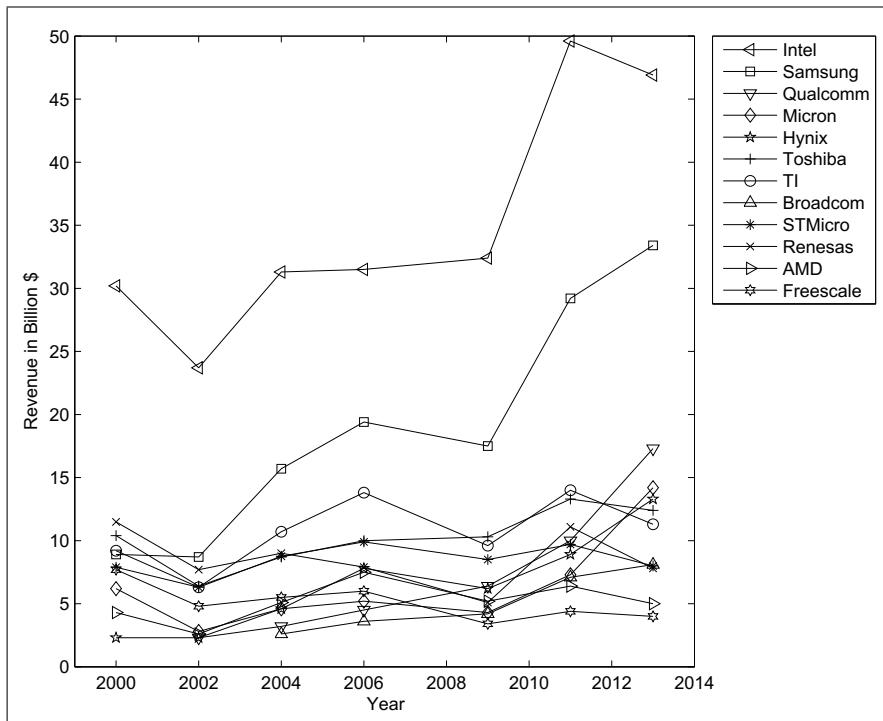


Fig. 9.2. Top semiconductor company revenue

592 million transistors. It includes 9 MB of L3 cache alone. The whole family of Intel processors is shown in Table 9.1.

Looking at the revenue of semiconductor companies it is still impressive that Intel has maintained its lead over many years mainly with just one product, the microprocessor. Other microprocessor-dominated companies like Texas Instruments, Motorola/Freescale or AMD have much lower revenue. Other top companies such as Samsung or Toshiba are dominated by memory technology, but still do not have Intel's revenue, which has been in the lead for many years now; see Fig. 9.2.

9.1.2 Brief History of RISC Microprocessors

The Intel architecture discussed in the last section is sometimes called a complex instruction set computer (CISC). Starting from the early CPUs, subsequent designs tried to be compatible, i.e., being able to run the same programs. As the bit width of data and programs expanded you can imagine that this compatibility came at a price: performance, although Moore's law allowed this CISC architecture to be quite successful by adding new components and features like numeric coprocessors, data and program caches,

MMX, and SSE instructions and the appropriate instructions that support these additions to improve performance. Intel µPs are characterized by having many instructions and supporting many addressing modes. The µP manuals are usually 700 or more pages thick.

Around 1980 research from CA University of Berkeley (Prof. Patterson), IBM (later called PowerPC) and at Stanford University (Prof. Hennessy, on what later become the microprocessor without interlocked pipeline stages, i.e., MIPS µP family) analyzed µPs and came to the conclusion that, from a performance standpoint, CISC machines had several problems. In Berkeley this new generation of µPs were called RISC-1 and RISC-2 since one of their most important feature was a limited number of instructions and addressing modes, leading to the name reduced instruction set computer (RISC).

Let us now briefly review some of the most important features by which (at least early) RISC and CISC machines could be characterized.

- The *instruction set* of a CISC machines is rich, while a RISC machine typically supports fewer than 100 instructions.
- The *word length* of RISC instructions is fixed, and typically 32 bits long. In a CISC machine the word length is variable. In Intel machines we find instructions from 1 to 15 bytes long.
- The *addressing modes* supported by a CISC machine are rich, while in a RISC machine only very few modes are supported. A typically RISC machine supports just immediate and register base addressing.
- The *operands* for ALU operations in a CISC machine can come from instruction words, registers, or memory. In a RISC machine no direct memory operands are used. Only memory load/store to or from registers is allowed and RISC machine are therefore called load/store architectures.
- In *subroutines* parameters and data are usually linked via a stack in CISC machines. A RISC machine has a substantial number of registers, which are used to link parameter and data to subroutines.

In the early 1990s it became apparent that neither RISC nor CISC architectures in their purest form were better for all applications. CISC machines, although still supporting many instruction and addressing modes, today take advantage of a larger number of CPU registers and deep pipelining. RISC machines today like the MIPS, PowerPC, SUN Sparc or DEC alpha, have hundreds of instructions, and some need multiple cycles and hardly fit the title reduced instruction set computer.

9.1.3 Brief History of PDSPs

In 1980 Intel [346] introduced the 2930, an analog signal processor that was used in control systems and included an ADC and DAC on chip¹ to implement the algorithms in a DSP-like fashion; see Fig. 9.3.

¹ We would call today such a µP a microcontroller, since I/O functions are integrated on the chip.

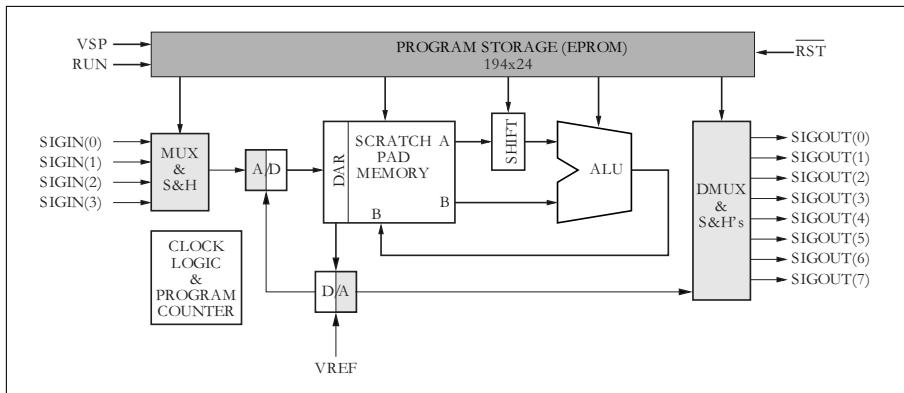


Fig. 9.3. The 2920 functional block diagram. [346]

The 2920 had a 40×25 bit scratch path memory, a 192-word program EPROM, a shifter, and an ALU and could implement multiplications with a series shift-and-add/subtracts.

Example 9.1: A multiplication in the 2920 with a constant $C = 1.88184_{10} = 1.1110000111_2$ was coded first in CSD as $C = 10.00\bar{1}000100\bar{1}_2$ and was then implemented with the following instructions:

```
ADD Y,X,L01;
SUB Y,X,R03;
ADD Y,X,R07;
SUB Y,X,R10;
```

where the last operand **Rxx** (**Lxx**) describes the right (left) shift by **xx** of the second operand before the adding or subtracting.

9.1

There are a few building block macrofunctions that can be used for the 2920, they are listed in the following table:

Function	# Instructions
Constant multiply	1–5
Variable multiply	10–26
Triangle generator	6–10
Threshold detector	2–4
Sine wave generator	8–12
Single real pole	2–6

although we will not call the 2920 a PDSP it shares same important features with the PDSP, which was improved with the first-generation PDSPs introduced at the start of the 1980s. The first-generation PDSPs like TI's TSM320C10 and NEC's μ PD 7720 were characterized by a *Harvard architecture*, i.e., the program and data were located in separate memory. The first

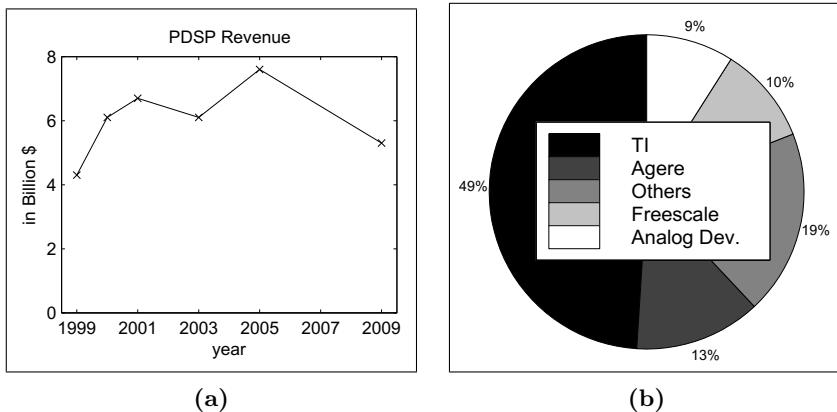


Fig. 9.4. PDSPs. (a) Processor revenue. (b) Market share. (Agere formerly Lucent/AT&T; Freescale formerly Motorola)

generation also had a hardwired multiplier that consumed most of the chip area and allowed a single multiply in one clock cycle. The second-generation PDSPs introduced around 1985 with TI's TMS 320C25, Motorola's MC56000, or the DSP16 from AT&T allowed a MAC operation in one clock cycle and zero-overhead loops were introduced. The third generation of PDSPs, with μ Ps like the TMS320C30, Motorola's 96002, and the DSP32C, introduced after 1988, now supported a single clock cycle 32-bit floating-point multiplication typically using the IEEE single-precision floating-point standard, see Sect. 2.2.3, p. 75. The forth generation with the TMS320C40 and TMS320C80 introduced around 1992 now included multi-core MACs. The newest and most powerful PDSPs introduced after 1997 like the TMS320C60x, Philips Trimedia, or Motorola Starcore, are very long instruction word (VLIW) machines. Other architectures in today's PDSPs are SIMD architectures (e.g., ADSP-2126x SHARC), superscalar machines (e.g., Renesas SH77xxx), matrix math engine (e.g., Intrinsity, FastMath), or a combination of these techniques. PDSPs architectures today are more diverse than ever.

PDSPs have done very well in the last 20 years, as can be seen from Fig. 9.4a and are expected to deliver three trillion instructions per second by 2010 [347]. The market share of PDSPs has not changed much over the years and is shown in Fig. 9.4b, with Texas Instruments (TI) leading the field by a large margin. Recently some PDSP cores have also become available for FPGA designs such as Cast Inc.'s (www.cast.com) version of the TI TMS32025 and Motorola/Freescale's 56000. We can assume that most of the core instructions and architecture features are supported, while some of the I/O units such as the timer, DMA, or UARTs included on the PDSPs are usually not implemented in the FPGA cores.

9.2 Instruction Set Design

The instruction set of a microprocessor (μ P) describes the collection of actions a μ P can perform. The designer usually asks first, which kind of *arithmetic* operation do I need in the applications for which I will use my μ P. For DSP application, for instance, special concern would be applied to support for (fast) add and multiply. A multiply done by a series of smaller shift-adds is probably not a good choice in heavy DSP processing. As well as these ALU operations we also need some *data move* instructions and some *program flow*-type instructions such as `branch` or `goto`.

The design of an instruction set also depends however on the underlying μ P architecture. Without considering the hardware elements we cannot fully define our instruction set. Because instruction set design is a complex task, it is a good idea to break up the development into several steps. We will proceed by answering the following questions:

- 1) What are the addressing modes the μ P supports?
- 2) What is the underlying data flow architecture, i.e., how many operands are involved in an instruction?
- 3) Where can we find each of these operands (e.g., register, memory, ports)?
- 4) What type of operations are supported?
- 5) Where can the next instruction be found?

9.2.1 Addressing Modes

Addressing modes describe how the operands for an operation are located. A CISC machine may support many different modes, while a design for performance like in RISC or PDSPs requires a limitation to the most often used addressing modes. Let us now have a look at the most frequently supported modes in RISC and PDSPs.

Implied addressing. In implied addressing operands come from or go to a location that is implicitly and not explicitly defined by the instruction; see Fig. 9.5. An example would be the ADD operation (no operands listed) used in a stack machine. All arithmetic operations in a stack machine are performed using the two top elements of the stack. Another example is the ZAC operation of a PDSP, which clears the accumulator in a TMS320 PDSP [348]. The following listing shows some examples of implied addressing for different microprocessors:

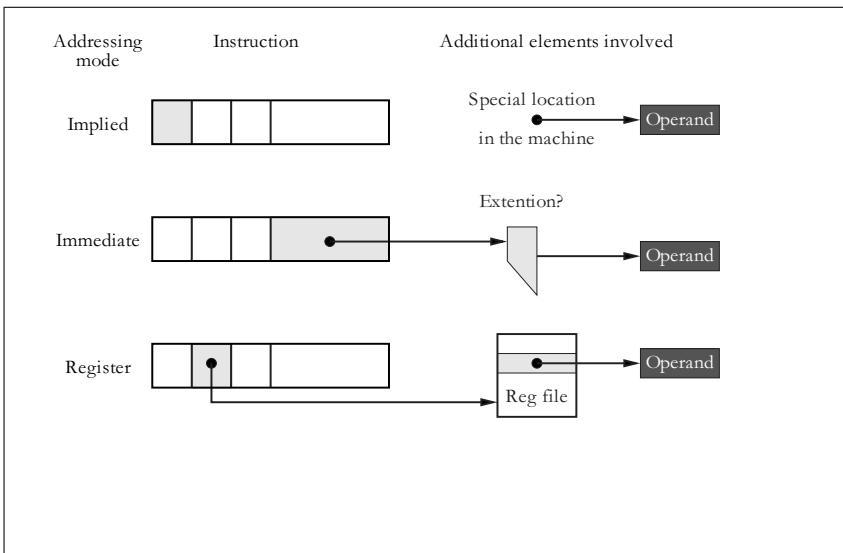


Fig. 9.5. Implied, immediate and register addressing

Instruction	Description	μ P
ZAT	Clear accumulator and T register.	TMS320C50
APAC	The contents of the P register is added to the accumulator register and replaces the accumulator register value.	TMS320C50
RET	The PC is loaded with the value of the ra register.	Nios II
BRET	Copies the b status into the status register and loads the PC with the ba register value.	Nios II

Immediate addressing. In the immediate addressing mode the operand (i.e., constant) is included in the instruction itself. This is shown in Fig. 9.5. A small problem arises due to the fact that the constant provided within one instruction word is usually shorter in terms of number of bits than the (full) data words used in the μ P. There are several approaches to solve this problem:

- a) Use a sign extension since most constants (like increments or loop counters) used in programs are small and do not require full length anyway. We should use sign extension (i.e., MSB is copied to the high word) rather than zero extension so that the value -1 is extended correctly (see the MPY -5 example below).

- b) Use two or more separate instructions to load the low and high parts of the constant one after the other and concatenate the two parts to a full-length word (see the LPH DAT0 example below).
- c) Use a double-length instruction format to access long constants. If we extend the default size by a second word, this usually provides enough bits to load a full-precision constant (see RPT #1111h example below).
- d) Use a (barrel) shift alignment of the operand that aligns the constant in the desired way (see ADD #11h,2 example below).

The following listing shows five examples for immediate addressing for different microprocessors. Examples 2-5 describe the extension method mentioned above.

Instruction	Description	μ P
CNTR=10;	Set the loop counter to 10.	ADSP
MPY -5	The 13-bit constant is sign extended to 16 bits and then multiplied by the TREG0 register and the result is stored in the P register.	TMS320C50
LPH DAT0	Load the upper half of the 32-bit product register with the data found in memory location DAT0.	TMS320C50
RPT #1111h	Repeat the next instruction $1111_{16} + 1 = 4370_{10}$ times. This is a two-word instruction and the constant is 16 bits long.	TMS320C50
ADD #11h,2	Add 11h shifted by two bits to the accumulator.	TMS320C50

To avoid having always two memory accesses, we can also combine method (a) sign extension and (b) high/low addressing. We then only need to make sure that the load high is done after the sign extension of the low word.

Register addressing. In the register addressing mode operands are accessed from registers within the CPU, and no external memory access takes place, see Fig. 9.5. The following listing shows some examples of register addressing for different microprocessors:

Instruction	Description	μ P
MR=MX0*MY0(RND)	Multiply the MX0 and MY0 registers with round and store in the MR register.	ADSP
SUB sA,sB	Subtract the sB from the sA register and store in the sA register.	PicoBlaze
XOR r6,r7,r8	Compute the logical exclusive XOR of the registers r7 and r8 and store the result in the register r6.	Nios II
LAR AR3,#05h	Load the auxiliary register AR3 with the value 5.	TMS320C50
OR %i1,%i2	OR the registers i1 and i2 and replace i1.	Nios
SWAP %g2	Swap the 16-bit half-word values in the 32-bit register g2 and put it back in g2.	Nios

Since in most machines register access is much faster and consumes less power than regular memory access this is a frequently used mode in RISC machines. In fact all arithmetic operations in a RISC machine are usually done with CPU registers only; memory access is only allowed via separate load/store operations.

Memory addressing. To access external memory direct, indirect, and combination modes are typically used. In the *direct* addressing mode part of the instruction word specifies the memory address to be accessed, see Fig. 9.6, and the **FETCH** example below. Here the same problem as for immediate addressing occurs: the bits provided in the instruction word to access the memory operand is too small to specify the full memory address. The full address length can be constructed by using an auxiliary register that may be ex- or implicitly specified. If the auxiliary register is added to the direct memory address this is called *based* addressing, see the **LDBU** example below. If the auxiliary register is just used to provide the missing MSB this is called *page-wise* addressing, since the auxiliary register allows us to specify a page within which we can access our data, see the **AND** example below. If we need to access data outside the page, we need to update the page pointer first. Since the register in the based addressing mode represents a full-length address we can use the register without a direct memory address. This is then called *indirect* addressing, see the **PM(I6,M6)** example below.

The following listing shows four examples for typical memory addressing modes for different microprocessors:

Instruction	Description	μ P
FETCH sX,ss	Read the scratch pad RAM location ss into register sX.	PicoBlaze
LDBU r6,100(r5)	Compute the sum of 100 and the register r5 and load the data from this address into the register r6.	Nios II
AND DAT16	Assuming that the 9-bit data page register points to page 4, this will access the data word from memory location $4 \times 128 + 16$ and perform an AND operation with the accumulator.	TMS320C50
PM(I6,M6)=AR	The AR register is stored in the program memory by using the register I6 as the address. After the memory access the address register I6 is incremented by the value from the modify register M6.	ADSP

Since the indirect address mode can usually only point to a limited number of index registers this usually shortens the instruction words and it is the most popular addressing mode in PDSPs. In RISC machines based addressing is preferred, since it allows easy access to an array by specifying the base and the array element via an offset, see the LDBU example above.

PDSP specific addressing modes. PDSPs are known to process DSP algorithms much more efficiently than standard GPP or RISC machines. We now want to discuss three addressing modes typical used in PDSPs that have shown to be particular helpful in DSP algorithms:

- Auto de/increment addressing
- Circular addressing
- Bitreversed addressing

The auto and circular addressing modes let us compute convolutions and correlations more efficiently, while bitreversed addressing is used in FFT and DCT algorithms.

The *auto increment* or *decrement* of the address pointer is a typical address modification that is used in PDSPs for convolutions or correlations. Since convolutions or correlations can be described by a repeated multiply-accumulate (MAC) operation, i.e., the inner (scalar) product of two vectors [see (3.2), p. 180] it can be argued that, after each MAC operation, an address modification increment or decrement depending on the ordering of the data and coefficient is necessary to update the address pointer for the

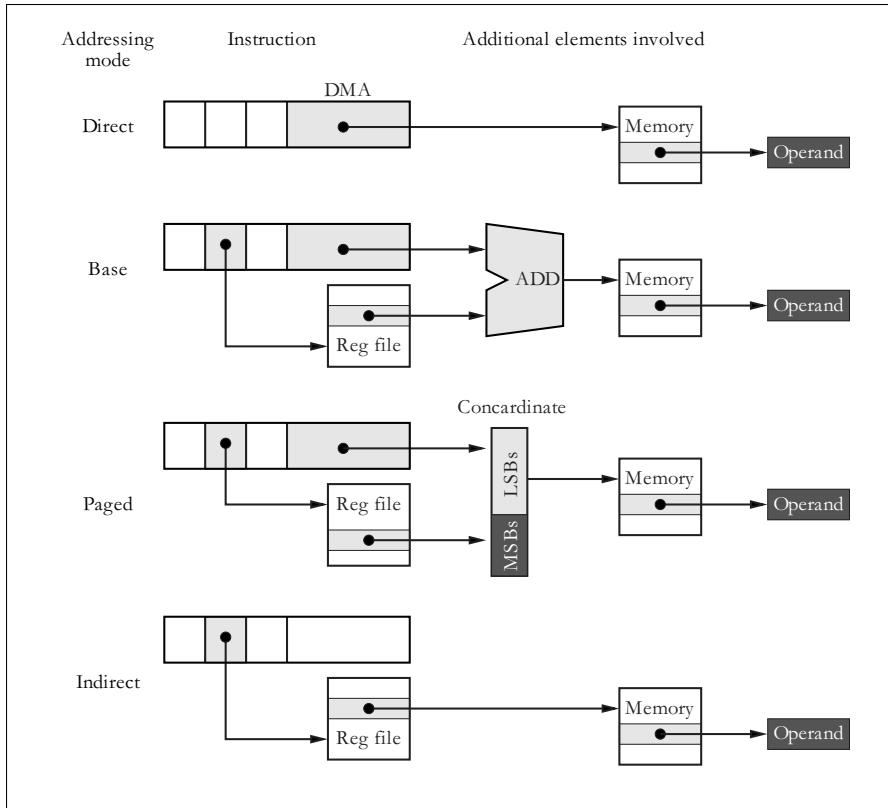


Fig. 9.6. Memory addressing: direct, based, paged, and indirect

next data/coefficient pair. After each memory access a de/increment of the data/coefficient read pointer is therefore performed. This allow us to design a single-cycle MAC – the address update is not done by the CPU as in RISC machines; a separate address register file is used for this purpose; see Fig. 9.7. An FIR filtering using the ADSP assembler coding can be done via the following few steps:

```
CNTR = 256;
MR=0, MX0=DM(I0,M1), MY0(I4,M5)
DO FIR UNTIL CE;
FIR: MR=MR+MX0*MY0(SS), MX0=DM(I0,M1), MY0=PM(I4,M5)
```

After initializing the loop counter CNTR and the pointer for data x coming from the data memory DM and y coming from the program memory PM, the DO UNTIL loop allows one to compute one MAC instruction and two data loads in each clock cycle. The data pointers I0 and I4 are then updated via M1 and M5, respectively. This example also shows another feature of PDSPs,

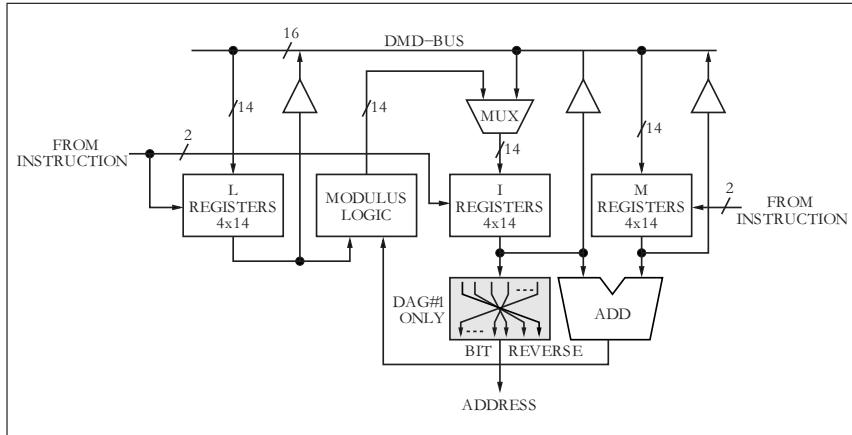


Fig. 9.7. Address generation in the ADSP programmable digital signal processor

the so-called zero-overhead loop. The update and loop counter check done at the end of the loop do not require any additional cycles as in GPP or RISC machines.

To motivate *circular addressing* let us revisit how data and coefficient typically are arranged in a continuous data processing scheme. In the first instance to compute $y[0]$ the data $x[k]$ and coefficient $f[k]$ are aligned as follows:

$f[L - 1]$	$f[L - 2]$	$f[L - 3]$	\dots	$f[1]$	$f[0]$
$x[0]$	$x[1]$	$x[2]$	\dots	$x[L - 2]$	$x[L - 1]$

where the coefficient and data in the same column are multiplied and all products are accumulated to compute $y[0]$. In the next step to compute $y[1]$ we need the following data:

$f[L - 1]$	$f[L - 2]$	$f[L - 3]$	\dots	$f[1]$	$f[0]$
$x[1]$	$x[2]$	$x[3]$	\dots	$x[L - 1]$	$x[L]$

We can solve this problem by shifting every data word after a MAC operation. The TMS320 family, for instance, provides such a MACD, i.e., a MAC plus data move instruction [349]. After N MACD operations the whole vector \mathbf{x} is shifted by one position. Alternatively we can look at the data that are used in the computation and we see that all data can stay in the same place; only the oldest element $x[0]$ needs to be replaced by $x[L]$ – the other coefficients keep their place. We therefore have the following memory arrangement for the \mathbf{x} data vector:

$x[L]$	$x[1]$	$x[2]$	\dots	$x[L - 2]$	$x[L - 1]$
--------	--------	--------	---------	------------	------------

If we now start our data pointer with $x[1]$ the processing works fine as before; only when we reach the end of the data buffer $x[L - 1]$ we do need to reset

Table 9.2. Properties of PDSPs useful in convolutions. (© Springer Press [5])

Vendor	Type	Accu bits	Super Harvard	Modulo address	Bit- re- verse	Hard- ware loops	MAC rate MHz
PDSPs 16×16-bit integer multiplier							
Analog Device	ADSP-2187	40	✓	✓	✓	✓	52
	ADSP-21csp01	40	✓	✓	—	—	50
Lucent	DSP1620	36	✓	—	✓	✓	120
Motorola	DSP56166	36	✓	✓	✓	✓	60
NEC	μPD77015	40	✓	✓	✓	✓	33
Texas Instruments	TMS320F206	32	✓	—	✓	✓	80
	TMS320C51	32	✓	✓	✓	✓	100
	TMS320C549	40	✓	✓	✓	✓	200
	TMS320C601	40	2 MAC	✓	✓	✓	200
	TMS320C80	32	4 MAC	✓	✓	✓	50
PDSPs 24×24-bit integer multiplier							
Motorola	DSP56011	56	✓	✓	✓	✓	80
	DSP56305	56	✓	✓	✓	✓	100
PDSPs 24×24-bit float multiplier							
Analog Device	SHARC 21061	80	✓	✓	✓	✓	40
Motorola	DSP96002	96	✓	✓	✓	✓	60
Texas Instruments	TMS320C31	40	✓	✓	✓	✓	60
	TMS320C40	40	✓	✓	✓	✓	60

the address pointer. Assuming the address buffer can be described by the following four parameters:

- L = buffer length
- I = current address
- M = modify value (signed)
- B = base address of buffer

we can describe the required address computation by the following equation:

$$\text{new address} = (I + M - B) \bmod L + B. \quad (9.1)$$

This is called circular addressing, since after each memory modification we check if the resulting address is still in the valid range. Figure 9.7 shows the address generator that is used by the ADSP PDSPs, using (9.1).

The third special addressing mode we find in PDSPs is used to simplify the radix-2 FFT. We have discussed in Chap. 6 that the input or output sequence in radix-2 FFTs appear in bitreverse order; see Fig. 6.14, p. 441. The bitreverse index computation usually take many clock cycles in software since the location of each bit must be reversed. PDSPs support this by a special addressing mode as the following example shows.

Example 9.2: The ADSP [350] and TMS320C50 [349] both support bitreverse addressing. Here is an assembler code example from the TMS320 family:

```
ADD * BR0-,8
```

The content of the INDX register is first subtracted from the current auxiliary register. Then a bitreverse of the address value is performed to locate the operand. The loaded data word is added after a shift of 8 bits to the accumulator.

9.2

Table 9.2 shows an overview of PDSPs and the supported addressing modes. All support auto de/increment. Circular buffers and bitreverse is also supported by most PDSPs.

9.2.2 Data Flow: Zero-, One-, Two- or Three-Address Design

A typical assembler coding of an instruction lists first the operation code followed by the operand(s). A typical ALU operation requires two operands and, if we also want to specify a separate result location, a natural way that makes assembler easy for the programmer would be to allow that the instruction word has an operation code followed by three operands. However, a three-operand choice can require a long instruction word. Assume our embedded FPGA memory has 1K words then at least 30 bits, not counting the operation code, are required in the direct addressing mode. In a modern CPU that address 4 GB requires a 32-bit address, three operands in direct addressing would require at least 96-bit instruction words. As a result limiting the number of operands will reduce the instruction word length and save resources. A zero-address or stack machine would be perfect in this regard. Another way would be to use a register file instead of direct memory access and only allow load/store of single operands as is typical in RISC machines. For a CPU with eight registers we would only need 9 bits to specify the three operands. But we then need extra operation code to load/store data memory data in the register file.

In the following sections we will discuss the implications the hardware and instruction sets when we allow zero to three operands in the instruction word.

Stack machine: a zero-address CPU. A zero-address machine, you may ask, how can this work? We need to recall from the addressing modes, see Sect. 9.2.1, p. 638, that operands can be specified implicitly in the instruction. For instance in a TI PDSP all products are stored in the product register P,

and this does not need to be specified in the multiply instruction since all multiply results will go there. Similarly in a zero-address or stack machine, all two-operand arithmetic operations are performed with the two top elements of a stack [351]. A stack by the way can be seen as a last-in first-out (LIFO). The element we put on the stack with the instruction PUSH will be the first that comes out when we use a POP operation. Let us briefly analyze how an expression like

$$d = 5 - a + b * c \quad (9.2)$$

is computed by a stack machine. The left side shows the instruction and the right side the contents of the stack with four entries. The top of the stack is to the left.

Instruction	Stack			
	TOP	2	3	4
PUSH 5	5	—	—	—
PUSH a	a	5	—	—
SUB	5 - a	—	—	—
PUSH b	b	5 - a	—	—
PUSH c	c	b	5 - a	—
MUL	$c \times b$	5 - a	—	—
ADD	$c \times b + 5 - a$	—	—	—
POP d	—	—	—	—

It can be seen that all arithmetic operations (ADD, SUB, MUL) use the implicitly specified operands top-of-stack and second-of stack, and are in fact zero-address operations. The memory operations PUSH and POP however require one operand.

The code for the stack machine is called postfix (or reverse Polish) operation, since first the operands are specified and then the operations. The standard arithmetic as in (9.2) is called infix notation, e.g., we have the two congruent representations:

$$\underbrace{5 - a + b * c}_{\text{Infix}} \longleftrightarrow \underbrace{5a - bc * +}_{\text{Postfix}} \quad (9.3)$$

In Exercises 9.8-9.11 (p. 730) some more examples of these two different arithmetic modes are shown. Some may recall that the postfix notation is exactly the same coding the HP41C pocket calculator requires. The HP41C too used a stack with four values. Figure 9.8a shows the machine architecture.

Accumulator machine: a one-address CPU. Let us now add a single accumulator to the CPU and use this accumulator both as the source for one operand and as the destination for the result. The arithmetic operations are of the form

$$\text{acc} \leftarrow \text{acc } \square \text{ op1}, \quad (9.4)$$

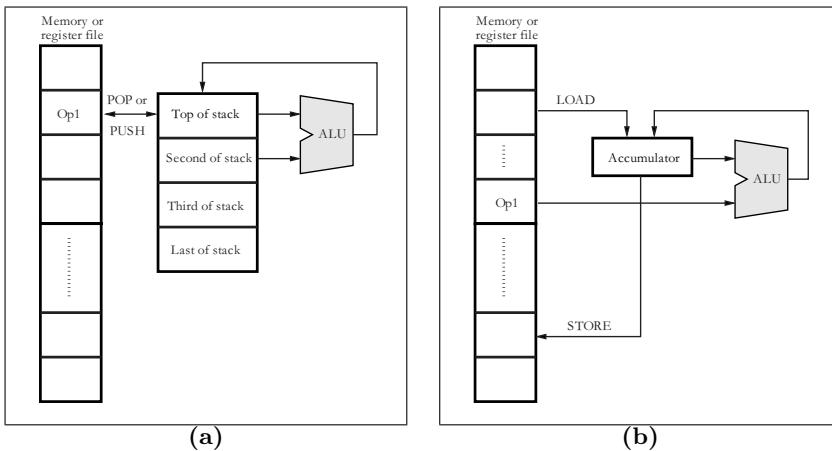


Fig. 9.8. (a) Stack CPU architecture. **(b)** Accumulator machine architecture

where \square describes an ALU operation like ADD, MUL, or AND. The underlying architecture of the TI TMS320 [348] family of PDSPs is of this type and is shown in Fig. 9.8b. In ADD or SUB operations, for instance, a single operand is specified. The example equation (9.2) from the last section would be coded in the TMS320C50 [349] assembler code as follows:

Instruction	Description
ZAP	; Clear accu and P register
ADD #5h	; Add 5 to the accu
SUB DAT1	; Subtract DAT1 from the accu
LT DAT2	; Load DAT2 in the T register
MPY DAT3	; Multiply T and DAT3 and store in P register
APAC	; Add the P register to the accu
SACL DAT4	; Store accu at the address DAT4

The example assumes that the variables **a-d** have been mapped to data memory words DAT1-DAT4. Comparing the stack machine with the accumulator machine we can make the following conclusions:

- The size of the instruction word has not changed, since the stack machine also requires POP and PUSH operations that include an operand
- The number of instructions to code an algebraic expression is not essentially reduced (seven for an accumulator machine; eight for a stack machine)

A more-substantial reduction in the number of instructions required to code an algebraic expression is expected when we use a two-operand machine, as discussed next.

The two-address CPU. In a two-address machine we have arithmetic operations that allows us to specify the two operands independently, and the destination operand is equal to the first operand, i.e., the operations are of the form

$$\text{op1} \leftarrow \text{op1} \square \text{op2}, \quad (9.5)$$

where \square describes an ALU operation like **SUB**, **DIV**, or **NAND**. The PicoBlaze from Xilinx [335,352] and the first generation Nios processor [353] from Altera use this kind of data flow, which is shown in Fig. 9.9a. The limitation to two operands allows in these cases the use of a 16-bit instruction word² format. The coding of our algebraic example equation (9.2) would be coded in assembler for the PicoBlaze as follows:

Instruction	Description
LOAD sD,sB	; Store register B in register D
MUL sD,sC	; Multiply D with register C
ADD sD,5	; Add 5 to D
SUB sD,sA	; Subtract A from D

In order to avoid an intermediate result for the product a rearrangement of the operation was necessary. Note that PicoBlaze does not have a separate **MUL** operation and the code is therefore for demonstration of the two-operand principle only. The PicoBlaze uses 16 registers, each 8 bits wide. With two operands and 8-bit constant values this allows us to fit the operation code and operands or constant in one 16-bit data word.

We can also see that two-operand coding reduces the number of operations essentially compared with stack or accumulator machines.

The three-address CPU. The three-address machine is the most flexible of all. The two operands and the destination operand can come or go into different registers or memory locations, i.e., the operations are of the form

$$\text{op1} \leftarrow \text{op2} \square \text{op3}. \quad (9.6)$$

Most modern RISC machine like the PowerPC, MicroBlaze or Nios II favor this type of coding [354–356]. The operands however are usually register operands or no more than one operand can come from data memory. The data flow is shown in Fig. 9.9b.

Programming in assembler language with the three-operand machine is a straightforward task. The coding of our arithmetic example equation (9.2) will look for a Nios II machine as follows:

² Some recent PicoBlaze coding now use 18-bit instruction words since this is the memory width of the Xilinx block RAMs [335,352].

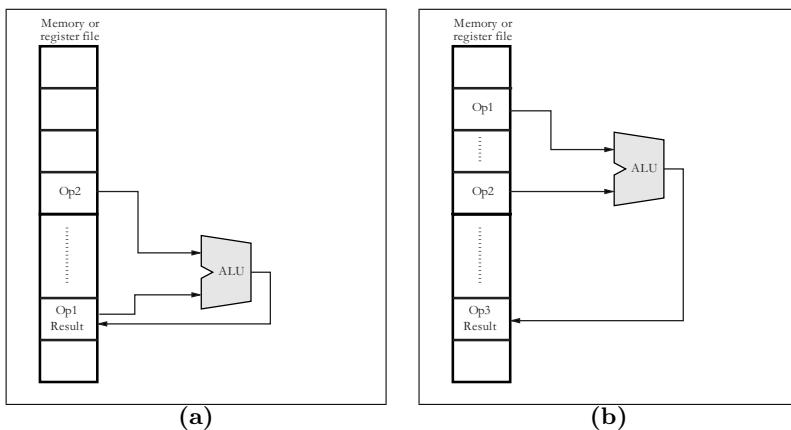


Fig. 9.9. (a) Two address CPU architecture. (b) Three address machine architecture

Instruction	Description
SUBI r4,r1,5	; Subtract 5 from r1 register and store in r4
MUL r5,r2,r3	; Multiply registers r2 and r3 and store in r5
ADD r4,r4,r5	; Add registers r4 and r5 and store in r4

assuming that the registers $r1-r4$ hold the values for the variables a through d . This is the shortest code of all four machines we have discussed so far. The price to pay is the larger instruction word. In terms of hardware implementation we will not see much difference between two- and three-operand machines, since the register files need separate multiplexer and demultiplexer anyway.

Comparison of Zero-, One-, Two- and Three-Address CPUs

Let us summarize our findings:

- The stack machine has the longest program and the shortest individual instructions.
- Even a stack machine needs a one-address instruction to access memory.
- The three-address machine has the shortest code but requires the largest number of bits per instruction.
- A register file can reduce the size of the instruction words. Typically in three-address machines two registers and one memory operand are allowed.
- A load/store machine only allows data moves between memory and registers. Any ALU operation is done with the register file.
- Most designs make the assumption that register access is faster than memory access. While this is true in CBIC or FPGAs that use external memory,

Table 9.3. Comparison of different design goal in zero- to three-operand CPUs

Goal	# of operands			
	0	1	2	3
Ease of assembler	worst	best
Simple C compiler	best	worst
# of code words	worst	best
Instruction length	best	worst
Range of immediate	worst	best
Fast operand fetch and decode	best	worst
Hardware size	best	worst

inside the FPGA register file access and embedded memory access times are in the same range, providing the option to realize the register file with embedded (three-port) memories.

The above finding seems unsatisfying. There seems no best choice and as a result each style has been used in practice as our coding examples show. The question then is: why hasn't one particular data flow type emerged as optimal? An answer to this question is not trivial since many factors, like ease of programming, size of code, speed of processing, and hardware requirements need to be considered. Let us compare the different designs based on this different design goals. The summary is shown in Table 9.3.

The *ease of assembler* coding is proportional to the complexity of the instruction. A three-address assembler code is much easier to read and code than the many PUSH and POP operations we find in stack machine assembler coding. The design of a *simple C compiler* on the other hand is much simpler for the stack machine since it easily employs the postfix operation that can be much more simply analyzed by a parser. Managing a register file in an efficient way is a very hard task for a compiler. The number of *code words* in arithmetic operation is much shorter for two- and three-address operation, since intermediate results can easily be computed. The *instruction length* is directly proportional to the number of operands. This can be simplified by using registers instead of direct memory access, but the instruction length still is much shorter with less operands. The size of the *immediate* operand that can be stored depends on the instruction length. With shorter instruction words the constants that can be embedded in the instructions are shorter and we may need multiple load or double word length instructions; see Fig. 9.6 (memory addressing). The operand *fetch and decode* is faster if fewer operands are involved. As a stack machine always uses the two top elements of a stack, no long MUX or DEMUX delays from register files occur. The *hardware size* mainly depends on the register file. A three-operand CPU has the highest requirements, a stack machine the smallest; ALU and control unit are similar in size.

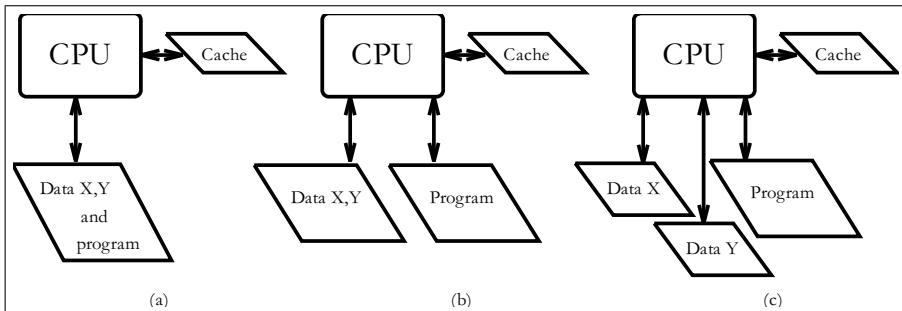


Fig. 9.10. Memory architectures. (a) Von-Neuman machine (GPP) [16, 17]. (b) Harvard architecture with separate program and data bus. (c) Super Harvard architecture with two data busses

In conclusion we can say that each particular architecture has its strengths and weaknesses and must also match the designer tools, skills, design goal in terms of size/speed/power, and development tools like the assembler, instruction set simulator, or C compiler.

9.2.3 Register File and Memory Architecture

In the early days of computers when memory was expensive *von Neuman* suggested a new highly celebrated innovation: to place the data and program in the same memory; see Fig. 9.10a. At that time computer programs were often hardwired in an FSM and only data memory used RAM. Nowadays the image is different: memory is cheap, but access speed is still for a typical RISC machine much slower than for CPU registers. In a three-address machine we therefore need to think about where the three operands should come from. Should all operands be allowed to come from main memory, or only two or one, or should we implement a *load/store architecture* that only allows single transfer between register and memory, but require that all ALU operations are done with CPU registers? The VAX PDP-11 is always quoted as the champion in this regard and allows multiple memory as well as multiple register operations. For an FPGA design we have the additional limitation that the number of instruction words is typically in the kilo range and the von Neuman approach is not a good choice. All the requirements for multiplexing data and program words would waste time and can be avoided if we use separate program and data memory. This is what is called a *Harvard architecture*, see Fig. 9.10b. For PDSPs designs it would be even better (think of an FIR filter application) if we can use three different memory ports: the coefficient and data come from two separate data memory locations x and y , while the accumulated results are held in CPU registers. The third memory is required for the program memory. Since many DSP algorithms are short, some PDSPs like the ADSP try to save the third bus by implementing a

small cache. After the first run through the loop the instructions are in the cache and the program memory can be used as a second data memory. This three-bus architecture is shown in Fig. 9.10c and is usually called a *super Harvard architecture*.

A GPP machine like Intel's Pentium or RISC machines usually use a memory hierarchy to provide the CPU with a continuous data stream but also allow one to use cheaper memory for the major data and programs. Such a memory hierarchy starts with very fast CPU registers, followed by level-1, level-2 data and/or program caches to main DRAM memory, and external media like CD-ROMs or tapes. The design of such a memory system is much more sophisticated than what we can design inside our FPGA.

From a hardware implementation standpoint the design of a CPU can be split into three main parts:

- Control path, i.e., a finite state machine,
- ALU
- Register file

of the three items, although not difficult to design, the register file seems to be the block with the highest cost when implemented with LEs. From these high implementation costs it appears that we need to compromise between more registers that make a μ P easy to use and the high implementation costs of a larger file, such as 32 registers. The following example shows the coding of a typical RISC register file.

Example 9.3: A RISC Register File

When designing a RISC register file we usually have a larger number of registers to implement. In order to avoid additional instructions for indirect addressing (offset zero) or to clear a register (both operands zero) or register move instructions, usually the first register is set permanently to zero. This may appear to be a large waste for a machine with few registers, but simplifies the assembler coding essential, as the following examples show:

Instruction	Description
ADD r3,r0,r0	; Set register r3 to zero.
ADD r4,r2,r0	; Move register r2 to register r4.
LDBU r5,100(r0)	Compute the sum of 100 and register r0=0 ; and load data from this address into register r5.

Note that the pseudo-instruction above only work under the assumption that the first register r0 is zero.

The following VHDL code³ shows the generic specification for a 16-register file with 8-bit width:

```
-- Description: This is a W x L bit register file.
-- First register is set to zero.
LIBRARY ieee;
```

³ The equivalent Verilog code `reg_file.v` for this example can be found in Appendix A on page 874. Synthesis results are shown in Appendix B on page 881.

```

USE ieee.std_logic_1164.ALL;

ENTITY reg_file IS
    GENERIC(W : INTEGER := 7; -- Bit width-1
            N : INTEGER := 15); -- Number of regs-1
    PORT(clk      : IN STD_LOGIC;      -- System clock
          reset    : IN STD_LOGIC;      -- Asynchronous reset
          reg_ena  : IN STD_LOGIC;      -- Write enable active 1
          data     : IN STD_LOGIC_VECTOR(W DOWNTO 0); -- Input
          rd       : IN INTEGER RANGE 0 TO N; -- Address for write
          rs       : IN INTEGER RANGE 0 TO N; -- 1. read address
          rt       : IN INTEGER RANGE 0 TO N; -- 2. read address
          s        : OUT STD_LOGIC_VECTOR(W DOWNTO 0); -- 1. data
          t        : OUT STD_LOGIC_VECTOR(W DOWNTO 0)); -- 2. data
END;

ARCHITECTURE fpga OF reg_file IS

SUBTYPE SLVW IS STD_LOGIC_VECTOR(W DOWNTO 0);
TYPE SLV_NxW IS ARRAY (0 TO N) OF SLVW;
SIGNAL r : SLV_NxW;

BEGIN

    MUX: PROCESS(clk, reset, data) -- Input mux inferring
    BEGIN
        IF reset = '1' THEN                      -- registers
            FOR K IN 0 TO N LOOP
                r(k) <= (OTHERS => '0');
            END LOOP;
        ELSIF rising_edge(clk) THEN
            IF reg_ena = '1' AND rd > 0 THEN
                r(rd) <= data;
            END IF;
        END IF;
    END PROCESS MUX;

    DEMUX: PROCESS (r, rs, rt) -- 2 output demux
    BEGIN
        IF rs > 0 THEN -- First source           -- without registers
            s <= r(rs);
        ELSE
            s <= (OTHERS => '0');
        END IF;
        IF rt > 0 THEN -- Second source
            t <= r(rt);
        ELSE
            t <= (OTHERS => '0');
        END IF;
    END PROCESS DEMUX;

END fpga;

```

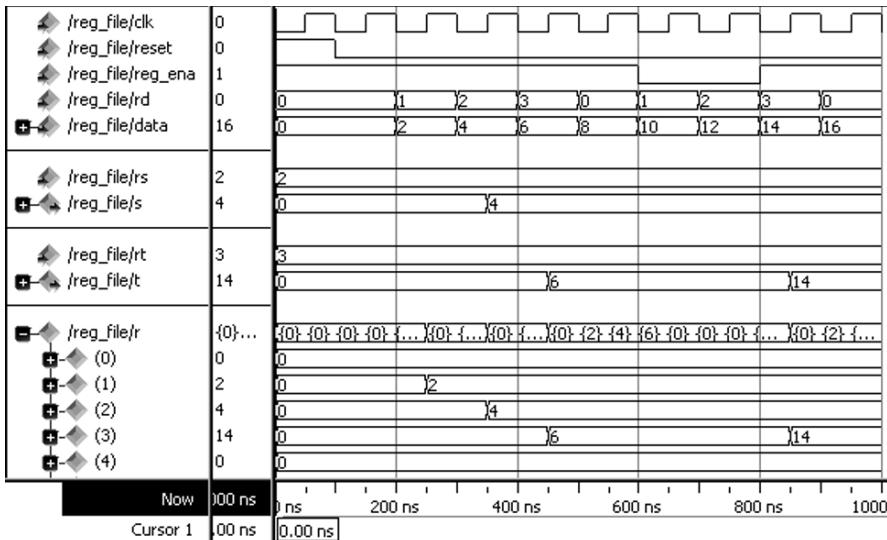


Fig. 9.11. VHDL simulation of the register file

The first process, **MUX**, is used to store the incoming data in the register file. Note that the register zero is not overwritten since it should be zero all the time. The second process, **DEMUX**, hosts the two decoders to read out the two operands for the ALU operations. Here again access to register 0 is answered with a value of zero. The design uses 226 LEs, no embedded multiplier, and no M9Ks. The **Fmax** registered performance cannot be measured since there is no register-to-register path.

We check the register file with the simulation shown in Fig. 9.11. The input **data** (address **rd**) is written continuously as data into the file. The output **s** is set via the **rs** to register 2, while output **t** is set to register 3 using **rt**. We notice that the register enable low signal between 600 and 800 ns means that register 2 is not overwritten with the data value 12. But for register 3 the enable is again high at 800 ns and the new value 14 is written into register 3, as can be seen from the **t** signal. The local variable **r** for the register file is shown last.

Figure 9.12 shows the LEs data for register number in the range 4 to 32 and bit width 8, 16, 24, and 32.

With Xilinx FPGAs it would also be possible to use the LEs as dual-port memory, see Chap. 1, Fig. 1.11, p. 21. With Altera FPGAs the only option to save the substantial number of LEs used for the register file would be to use a three-port memory, or two embedded dual-port memory blocks as the register file. We would write the same data in both memories and can read the two sources from the other port of the memory. This principle has been used in the Nios µP and can greatly reduce the LE count. We may then only use the lower 16 or 32 registers or offer (as it is done in the Nios µP) a window

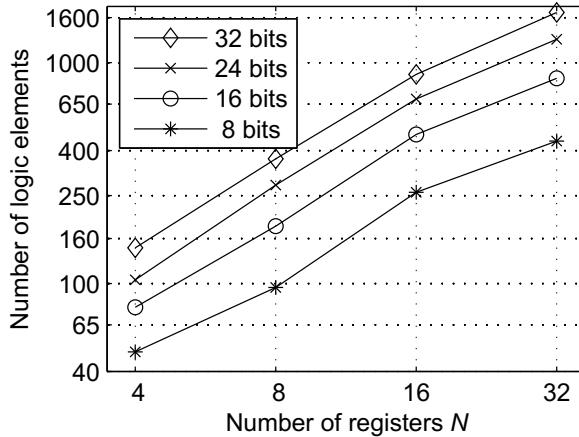


Fig. 9.12. LEs for different register file configurations

of registers. The window can be moved, for instance, in a subroutine call and the basic register do not need to be saved on a stack or in memory as would otherwise be necessary.

From the timing requirement however we now have the problem that BlockRAMs are synchronous memory blocks, and we cannot load and store memory addresses and data with the same clock edge from both ports, i.e., replacing the same register value using the current demultiplexer value cannot be done with the same clock edge. But we can use the rising edge to specify the operand address to be loaded and then use the falling edge to store the new value and set the write enable.

9.2.4 Operation Support

Most machines have at least one instruction out of the three categories: arithmetic/logic unit (ALU), data move, and program control. Let us in the following briefly review some typical examples from each category. The underlying data type is usually a multiple of bytes, i.e., 8, 16, or 32 bits of integer data type; some more-sophisticated processors use a 32- or 64-bit IEEE floating-point data type, see Sect. 2.2.3, p. 75.

ALU instructions. ALU instruction include arithmetic, logic, and shift operations. Typical supported arithmetic instructions for two operands are addition (**ADD**), subtraction (**SUB**), multiply (**MUL**) or multiply-and-accumulate (**MAC**). For a single operand, absolute (**ABS**) and sign inversion (**NEG**) are part of a minimum set. Division operation is typically done by a series of shift-subtract-compare instructions since an array divider can be quite large; see Fig. 2.27, p. 107.

The shift operation is useful since in b -bit integer arithmetic a bit growth to $2 \times b$ occurs after each multiplication. The shifter may be implicit as in the TMS320 PDSP from TI or provided as separate instructions. Logical and arithmetic (i.e., correct sign extension) as well as rotations are typically supported. In a block floating-point data format exponent detection (i.e., determining the number of sign bits) is also a required operation.

The following listing shows arithmetic and shift operations for different microprocessors:

Instruction	Description	μ P
ADD *,8,AR3	The * indicates that the auxiliary memory point ARP points to one of the eight address registers that is used for the memory access. The word from that location is left shifted by eight bits before being added to the accumulator. After the instruction the ARP points to AR3, which is used for the next instruction.	TMS320C50
MACD Coeff, Y	Multiply the coefficient and Y and store the result in the product register P. Then move the register Y by one location.	TMS320C50
NABS r3, r4	Store the negative absolute value of r4 in r3.	PowerPC
DIV r3,r2,r1	This instruction divides the register r2 by r1 and stores the quotient in the register r3.	Nios II
SR=SRr OR ASHIFT 5	Right shift the SR register by five bits and use sign extension.	ADSP

Although logic operations are less often used in basic DSP algorithms like filtering or FFT, some more-complex systems that use cryptography or error correction algorithms need basic logic operations such as AND, OR, and NOT. For error correction EXOR and EQUIV are also useful. If the instruction number is critical we can also use a single NAND or NOR operation and all other Boolean operations can be derived from these universal functions; see Exercise 1.1, p. 47.

Data move instructions. Due to the large address space and performance concerns most machines are closer to the typical RISC load/store architecture than the universal approach of the VAX PDP-11 that allows all operands of an instruction to come from memory. In the load/store philosophy we only allow data move instructions between memory and CPU registers, or different

registers – a memory location can *not* be part of an ALU operation. In PDSP designs a slightly different approach is taken. Most data access is done with indirect addressing, since a typical PDSP like the ADSP or TMS320 has separate memory address generation units that allow auto de/increment and modulo addressing; see Fig. 9.7, p. 644. These address computations are done in parallel to the CPU and do not require additional CPU clock cycles.

The following listing shows data move instructions for different microprocessors:

Instruction	Description	μ P
st [%fp], %g1	Store register g1 at the memory location specified in the fp register.	Nios
LWZ R5, DMA	Move the 32-bit data from the memory location specified by DMA to register R5	PowerPC
MX0=DM(I2, M1)	Load from data memory into register MX0 the word pointed to by the address register I2 and post-increment I2 by M1.	ADSP
IN STAT, DA3	Read the word from the peripheral on port address 3 and store the data in the new location STAT.	TMS320

Program flow instructions. Under control flow we group instructions that allow us to implement loops, call subroutines, or jump to a specific program location. We may also set the μ P to idle, waiting for an interrupt to occur, which indicates new data arrival that need to be processed.

One specific type of hardware support in PDSPs that is worth mentioning is the so-called zero-overhead loops. Usually at the end of a loop the μ P decrements the loop counter and checks if the end of the loop is reached. If not, the program flow continues at the begin of the loop. This check would require about four instructions, and with typical PDSP algorithms (e.g., FIR filter) with a loop length of instruction 1, i.e., a single MAC, 80% of the time would be spent on the loop control. The loops in PDSP are in fact so short that the TMS320C10 provides a RPT #imm instruction such that the next instruction is repeated #imm+1 times. Newer PDSPs like the ADSP or TMS320C50 also allow longer loops and nested loops of several levels. In most RISC machine applications the loops are usually not as short as for PDSPs and the loop overhead is not so critical. In addition RISC machines use delay branch slots to avoid NOPs in pipeline machines.

The following listing shows program flow instructions for different microprocessors:

Instruction	Description	μ P
CALL FIR	Call the subroutine that starts at the label FIR.	TMS32010
BUN r1	Branch to the location stored in the register r1 if in the previous floating-point operation one or more values were NaNs.	PowerPC
RET	The end of a subroutine is reached and the PC is loaded with the value stored in the register ra.	Nios II
RPT #7h	Repeat the next instruction 7 + 1 = 8 times. This is a one-word instruction due to the small constant value.	TMS320C50
CNTR=10; DO L UNTIL CE;	Repeat the loop from the next instruction on up to the label L until the counter CNTR expires.	ADSP

Additional hardware logic is provided by PDSPs to enable these short loops without requiring additional clock cycles at the end of the loop. The initialization of zero-overhead loops usually consists of specifying the number of loops and the end-of-loop label or the number of instructions in the loop; see the ADSP example above. Concurrently to the operation execution the control unit checks if the next instruction is still in the loop range, otherwise it loads the next instruction into the instruction register and continues with the instruction from the start of the loop. From the overview in Table 9.2, p. 645 it can be concluded that all second-generation PSDPs support this feature.

9.2.5 Next Operation Location

In theory we can simplify the next operation computation by providing a fourth operand that includes the address of the next instruction word. But since almost all instructions are executed one after the other (except for jump-type instructions), this is mainly redundant information and we find no commercial microprocessor today that uses this concept.

Only if we design an ultimate RISC machine (see Exercise 9.12, p. 730) that contains only one instruction we do need to include the next address or (better) the offset compared to the current instruction in the instruction word [357].

9.3 Software Tools

According to Altera's Nios online net seminar [358] one of the main reasons why Altera's Nios development systems have been a huge success⁴ is based on the fact that, besides a fully functional microprocessor, also all the necessary software tools including a GCC-based C compiler is generated at the same time when the IP block parametrization takes place. You can find many free µP cores on the web, see for instance:

- <http://www.opencores.org/> OPENCORES.ORG
- <http://www.free-ip.com/> The free IP project
- <http://www.fpgacpu.org/> FPGA CPU

but most of them lack a full set of development tools and are therefore less useful. A set of development tools (best case) should include

- Assembler, linker, and loader/basic terminal program
- Instruction set simulator
- C compiler

Figure 9.13 explains the different levels of abstraction in the development tools. In the following we will briefly describe the main programs used to develop these tools. You may also consider using the language for instruction set architecture (LISA) system originally developed at ISS, RWTH Aachen [359], and now a commercial product of Synopsys Inc., which automatically generates an assembler and instruction set simulator, and the C compiler with a few additional specifications in a semiautomatic way. In Sect. 9.5.2 (p. 706) we will review this type of design flow.

Writing a compiler can be a time-consuming project. A good C compiler, for instance, requires up to 50 man-years of work [360–362].

Nowadays we can benefit from the program developed in the GNU project that provides several useful utilities that speed up compiler development:

- The GNU tool **Flex** [363] is a scanner or lexical analyzer that recognizes patterns in text, similar to what the UNIX utility **grep** or the line editor **sed** can do for single pattern.
- The GNU tool **Bison** [364] a, YACC-compatible parser generator [365] allows us to describe a grammar in Bakus–Naur form (BNF), which can initiate actions if expressions are found in the text.
- For the GNU C compiler **gcc** we can take advantage of the tutorial written by R. Stallman [366] to adapt the C compiler to the actual µP we have or plan to build.

All three tools are freely available under the terms of the GNU public licence and for all three tools we have included documentation on the book CD under the **uP** folder, which also includes many useful examples.

⁴ 10,000 systems were sold in the first four years after introduction.

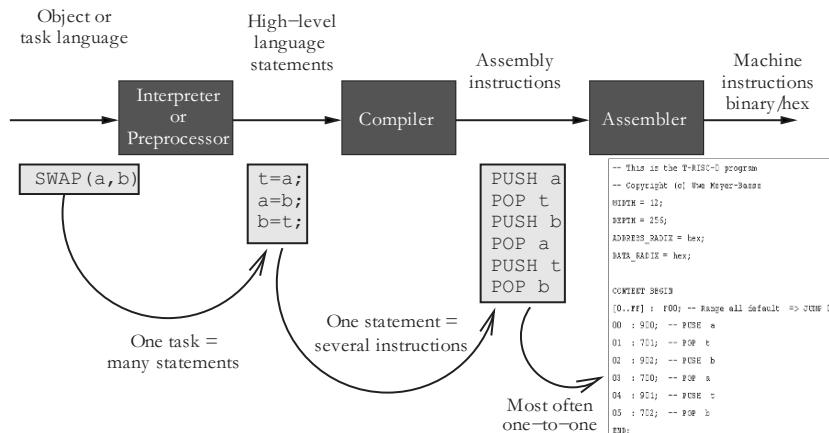


Fig. 9.13. Models and tools in programming

9.3.1 Lexical Analysis

A program that can recognize lexical patterns in a text is called a *scanner*. **Flex**, compatible with the original AT&T Lex, is a tool that can generate such a scanner [367]. **Flex** uses an input file (usual extension `*.l`) and produces C source code that can then be compiled on the same or a different system. A typical scenario is that you generate the parser under UNIX or Linux with the GNU tools and, since most Altera tools run on a PC, we compile the scanner under MS-DOS so that we can use it together with the Quartus II software. The default UNIX file that is produced by **Flex** uses a `lex.yy.c` file name. We can change this by using the option `-oNAME.C` to generate the output `NAME.C` instead. Note that there is no space between `-o` and the new name under UNIX. Assume we have a **Flex** input file `simple.l` then we use the two steps:

```
flex -osimple.c simple.l
gcc -o simple.exe simple.c
```

to generate a scanner called `simple.exe` under UNIX. Even a very short input file produces about 1,500 lines of C-code and has a size of 35 KB. We can already see from these data the great help this utility can be. We can also FTP the C-code `simple.c` to an MS-DOS PC and then compile it with a C compiler of our choice. The question now is how do we specify the pattern in **Flex** for our scanner, i.e., how does a typical **Flex** input file look? Let us first have a look at the formal arrangement in the **Flex** input file. The file consists of three parts:

`%{`

```
C header and defines come here
%}
definitions ...
%%
rules ...
%%
user C code ...
```

The three sections are separated by two %% symbols. Here is a short example of an input file.

```
/* A simple flex example */
%{
/* C-header and definitions */
#include <stdlib.h> /* needed for malloc, exit etc */
#define YY_MAIN 1
%}
%%
.|\n          ECHO; /* Rule section */
%%
/* User code here */
int yywrap(void) { return 1; }
```

The YY_MAIN define is used in case you want **Flex** to provide the main routine for the scanner. We could also have provided a basic main routine as follows:

```
main() { lex(); }
```

The most important part is the rule section. There we specify the pattern followed by the actions. The pattern . is any character except new line, and \n is new line. The bar | stands for the or combination. You can see that most coding has a heavy C-code flavor. The associated action ECHO will forward each character to standard output. So our scanner works similarly to the **more**, **type**, or **cat** utility you may have used before. Note that **Flex** is column sensitive. Only patterns in the rule section are allowed to start in the first column; not even a comment is allowed to start here. Between the pattern and the actions, or between multiple actions combined in parenthesis {}, a space is needed.

We have already discussed two special symbols used by **Flex**: the dot . that describes any character and the new line symbol \n. Table 9.4 shows the most often used symbols. Note that these are the same kinds of symbols used by the utility **grep** or the line editor **sed** to specify regular expressions. Here are some examples of how to specify a pattern:

Pattern	Matches
a	the character a.
a{1,3}	One to three a's, i.e., a aa aaa.
a b c	any single character from a, b, or c.
[a-c]	any single character from a, b, or c, i.e., a b c.
ab*	a and zero or more b's, i.e., a ab abb abbb...
ab+	a and one or more b's, i.e., ab abb abbb...
a\+b	string a+b .
[\t\n]+	one or more space, tab or new lines.
^L	Begin of line must be an L.
[^a-b]	any character except a, b, or c.

Using these pattern we now build a more-useful scanner that performs a lexical analysis of a VHDL file and reports the types of items he finds. Here is the Flex file vhdllex.l

```
/* Lexical analysis for a toy VHDL-like language */

%{
#include <stdio.h>
#include <stdlib.h>
%}
DIGIT          [0-9]
ID             [a-z] [a-zA-Z_]* 
ASSIGNMENT     [(<=)|(:=)] 
GENERIC        [A-Z]
DELIMITER      [;,)(':]
COMMENT        "--" [^\n]* 
LABEL          [a-zA-Z] [a-zA-Z_0-9]* [:]
%%
{DIGIT}+        { printf( "An integer: %s (%d)\n", yytext,
                      atoi( yytext ) ); }
IN|OUT|ENTITY|IS|END|PORT|ARCHITECTURE|OF|WAIT|UNTIL {
               printf( "A keyword: %s\n", yytext ); }
BEGIN|PROCESS { printf( "A keyword: %s\n", yytext ); }

{ID}            printf( "An identifier: %s\n", yytext );
"<="           printf( "An assignment: %s\n", yytext );
"="            printf( "Equal condition: %s\n", yytext );
{DELIMITER}    printf( "A delimiter: %s\n", yytext );
{LABEL}         printf( "A label: %s\n", yytext );

"+|-*|"/"      printf( "An operator: %s\n", yytext );
{COMMENT}       printf( "A comment: %s\n", yytext );
[ \t\n]+        /* eat up whitespace */
```

Table 9.4. Special symbols used by Flex

Symbol	Meaning
.	Any single character except new line
\n	New line
*	Zero or more copies of the preceding expression
+	One or more copies of the preceding expression
?	Zero or one of the preceding expression
^	Begin of line or negated character class
\$	End of line symbol
	Alternate, i.e., or expressions
()	Group of expressions
"+"	Literal use of expression within quotes
[]	Character class
{}	How many times an expression is used
\	Escape sequence to use a special symbol as a character only

```
.
    printf( "Unrecognized character: %s\n", yytext );
%%

int yywrap(void) { return 1; }

main( argc, argv )
int argc;
char **argv;
{
    ++argv, --argc; /* skip over program name */
    if ( argc > 0 )
        yyin = fopen( argv[0], "r" );
    else
        yyin = stdin;
    yylex();
}
```

We compile the files with the following step under UNIX:

```
flex -ovhdlex.c vhdlex.l
gcc -o vhdlex.exe vhdlex.c
```

Assume we have the following small VHDL example:

```
ENTITY d_ff IS -- Example flip-flop PORT(clk, d      :IN bit;
                                         q      :OUT bit);
END;

ARCHITECTURE fpga OF d_ff IS BEGIN P1: PROCESS (clk)
```

```

BEGIN
    WAIT UNTIL clk='1'; --> gives always FF
        q <= d;
    END PROCESS;
END fpga;

```

then calling our scanner with `vhdlex.exe < d_ff.vhd` will produce the following output:

```

A keyword: ENTITY
An identifier: d_ff
A keyword: IS
A comment: -- Example flip-flop
A keyword: PORT
A delimiter: (
An identifier: clk
A delimiter: ,
An identifier: d
A delimiter: :
A keyword: IN
An identifier: bit
A delimiter: ;
An identifier: q
A delimiter: :
A keyword: OUT
An identifier: bit
A delimiter: )
A delimiter: ;
A keyword: END
A delimiter: ;
A keyword: ARCHITECTURE
An identifier: fpga
A keyword: OF
An identifier: d_ff
...

```

After the two introductory examples we can now take on a more-challenging task. Let us build an `asm2mif` converter that reads in assembler code and outputs an MIF file that can be loaded into the block memory as used by the Quartus II software. To keep things simple let us use the assembler code of a stack machine with the following 16 operations (sorted by their operation code):

```

ADD, NEG, SUB, OPAND, OPOR, INV, MUL, POP,
PUSHI, PUSH, SCAN, PRINT, CNE, CEQ, CJP, JMP

```

Since we should also allow forward referencing labels in the assembler code we need to have a two-pass analysis. In the first pass we make a list of all

variables and labels and their code lines. We also assign a memory location to variables when we find one. In the second run we can then translate our assembler code line-by-line into MIF code. We start with the MIF file header that includes the data formats and then each line will have a address, an operation and possibly an operand. We can display the original assembler code at the end of each line by preceding -- comment symbols as in VHDL. Here is the Flex input file for our two-pass scanner:

```
/* Scanner for assembler to MIF file converter */
%{
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <errno.h>
#include <stdlib.h>
#include <time.h>
#include <ctype.h>
#define DEBUG 0
int state =0; /* end of line prints out IW */
int  ict =0; /* number of instructions */
int  vcount =0; /* number of variables */
int  pp =1; /* preprocessor flag */
char opis[6], lblis[4], immis[4];
struct inst {int adr; char opc; int imm; char *txt;} iw;
struct init {char *name; char code;} op_table[20] = {
    "ADD" , '0', "NEG" , '1', "SUB" , '2',
    "OPAND" , '3', "OPOR" , '4', "INV" , '5',
    "MUL" , '6', "POP" , '7', "PUSHI" , '8',
    "PUSH" , '9', "SCAN" , 'a', "PRINT" , 'b',
    "CNE" , 'c', "CEQ" , 'd', "CJP" , 'e',
    "JMP" , 'f', 0,0 };
FILE *fid;
int add_symbol(int value, char *symbol);
int lookup_symbol(char *symbol);
void list_symbols();
void conv2hex(int value, int Width);
char lookup_opc(char *opc);
%}
DIGIT          [0-9]
VAR            [a-z][a-z0-9_]*
COMMENT        "--[^\\n]*"
LABEL          L[0-9]+[:]
GOTO           L[0-9]+
%%
\n           {if (pp) printf( "-- end of line \n");}
```

```

        else { if ((state==2) && (pp==0))
/* print out an instruction at end of line */
{conv2hex(iw.adr,8); printf(" : %c",iw.opc);
    conv2hex(iw.imm,8);
    printf("; -- %s %s\n",opis,immis); }
    state=0;iw.imm=0;
    }}
{DIGIT}+ { if (pp) printf( "-- An integer: %s (%d)\n",
                           yytext, atoi( yytext ) );
            else {iw.imm=atoi( yytext ); state=2;
                   strcpy(immis,yytext);}}
POP|PUSH|PUSHI|CJP|JMP {
    if (pp)
        printf( "-- %d) Instruction with operand: %s\n",
               icount++, yytext);
    else {state=1; iw.adr=icount++;
           iw.opc=lookup_opc(yytext);}}
CNE|CEQ|SCAN|PRINT|ADD|NEG|SUB|OPAND|OPOR|INV|MUL {
    if (pp) printf( "-- %d) ALU Instruction: %s\n",
                   icount++, yytext );
    else { state=2; iw.opc=lookup_opc(yytext);
           iw.adr=icount++; strcpy(immis," ");}
{VAR} { if (pp) {printf( "-- An identifier: %s\n",
                           yytext ); add_symbol(vcount, yytext);}
        else {state=2;iw.imm=lookup_symbol(yytext);
              strcpy(immis,yytext);}}
{LABEL} { if (pp) {printf( "-- A label: %s length=%d
                           Icount=%d\n", yytext , yyleng, icount);
                  add_symbol(icount, yytext);}}
{GOTO} {if (pp) printf( "-- A goto label: %s\n", yytext );
         else {state=2;sprintf(lblis,"%s:",yytext);
               iw.imm=lookup_symbol(lblis);strcpy(immis,yytext);}}
{COMMENT} {if (pp) printf( "-- A comment: %s\n", yytext );
[ \t]+ /* eat up whitespace */
.     printf( "Unrecognized character: %s\n", yytext );
%%

int yywrap(void) { return 1; }

int main( argc, argv )
int argc;
char **argv;
{
    ++argv, --argc; /* skip over program name */
}

```

```

if ( argc > 0 )
    yyin = fopen( argv[0], "r" );
else
{ printf("No input file -> EXIT\n"); exit(1);}
printf("--- First path though file ---\n");
yylex();
if (yyin != NULL) fclose(yyin);
pp=0;
printf("\n-- This is the T-RISC program with ");
printf("%d lines and %d variables\n",icount,vcount);
icount=0;
printf("-- for the book DSP with FPGAs\n");
printf("-- Copyright (c) Uwe Meyer-Baese\n");
printf("-- WIDTH = 12; DEPTH = 256;\n");
if (DEBUG) list_symbols();
printf("ADDRESS_RADIX = hex; DATA_RADIX = hex;\n\n");
printf("CONTENT BEGIN\n");
printf("[0..FF] : FOO; -- ");
printf("Set all address from 0 to 255 => JUMP 0\n");
if (DEBUG) printf("--- Second path through file ---\n");
yyin = fopen( argv[0], "r" );
yylex();
printf("END;\n");
}

/* define a linked list of symbols */
struct symbol { char *symbol_name; int symbol_value;
               struct symbol *next; };

struct symbol *symbol_list; /*first element in symbol list*/

extern void *malloc();

int add_symbol(int value, char *symbol)
{
    struct symbol *wp;

    if(lookup_symbol(symbol) >= 0 ) {
printf("-- Warning: symbol %s already defined \n", symbol);
        return 0;
    }
    wp = (struct symbol *) malloc(sizeof(struct symbol));
    wp->next = symbol_list;
    wp->symbol_name = (char *) malloc(strlen(symbol)+1);
}

```

```
strcpy(wp->symbol_name, symbol);
    if (symbol[0]!='L') vcount++;
wp->symbol_value = value;
symbol_list = wp;
return 1; /* it worked */
}

int lookup_symbol(char *symbol)
{
    struct symbol *wp = symbol_list;
    for(; wp; wp = wp->next) {
        if(strcmp(wp->symbol_name, symbol) == 0)
        {if (DEBUG)
            printf("-- Found symbol %s value is: %d\n",
                   symbol, wp->symbol_value);
         return wp->symbol_value;}
    }
    if (DEBUG) printf("-- Symbol %s not found!!\n",symbol);
    return -1; /* not found */
}

char lookup_opc(char *opc)
{ int k;
strcpy(opis,opc);
for (k=0; op_table[k].name != 0; k++)
    if (strcmp(opc,op_table[k].name) == 0)
        return (op_table[k].code);
printf("***** Ups, no opcode for: %s --> exit \n",opc);
exit(1);
}

void list_symbols()
{
    struct symbol *wp = symbol_list;
    printf(" --- Print the Symbol list: ---\n");
    for(; wp; wp = wp->next)
        if (wp->symbol_name[0]=='L') {
printf("-- Label      : %s  line = %d\n",
               wp->symbol_name, wp->symbol_value);
        } else {
printf("-- Variable   : %s  memory @ %d\n",
               wp->symbol_name, wp->symbol_value);
        }
}
```

```

}

***** CONV_STD_LOGIC_VECTOR(value, bits) *****/
void
conv2hex(int value, int Width)
{
    int          W, k, t;
    extern FILE *fid;
    t = value;
    for (k = Width - 4; k >= 0; k-=4) {
        W = (t >> k) % 16; printf( "%1x", W);
    }
}

```

The variable pp is used to decide if the preprocessing phase or the second phase is running. Labels and variables are stored in a symbol table using the functions `add_symbol` and `lookup_symbol`. For labels we store the instruction line the label occurs, while for variables we assign a running number as we go through the code. An output of the symbol table for two labels and two variables will look as follows:

```

...
--- Print the Symbol list: ---
-- Label    : L01: line = 17
-- Label    : L00: line = 4
-- Variable : k  memory @ 1
-- Variable : x  memory @ 0
...

```

This will be displayed in the debug mode (set `#define DEBUG 1`) of the scanner, to display the symbol list. Here are the UNIX instructions to compile and run the code:

```

flex -oasm2mif.c asm2mif.l
gcc -o asm2mif.exe asm2mif.c
asm2mif.exe factorial.asm

```

The factorial program `factorial.asm` for the stack machine looks as follows:

```

PUSHI   1
POP     x
SCAN
POP     k
L00:   PUSH   k
       PUSHI  1
       CNE
       CJP    L01

```

```

PUSH    x
PUSH    k
MUL
POP    x
PUSH    k
PUSHI   1
SUB
POP    k
JMP    L00
L01:   PUSH    x
        PRINT

```

The output generated by `asm2mif` is shown next.

```

-- This is the T-RISC program with 19 lines and 2 variables
-- for the book DSP with FPGAs
-- Copyright (c) Uwe Meyer-Baese
WIDTH = 12;
DEPTH = 256;
ADDRESS_RADIX = hex;
DATA_RADIX = hex;

CONTENT BEGIN
[0..FF] : FOO; -- Set address from 0 to 255 => JUMP 0
00 : 801; -- PUSHI 1
01 : 700; -- POP x
02 : a00; -- SCAN
03 : 701; -- POP k
04 : 901; -- PUSH k
05 : 801; -- PUSHI 1
06 : c00; -- CNE
07 : e11; -- CJP L01
08 : 900; -- PUSH x
09 : 901; -- PUSH k
0a : 600; -- MUL
0b : 700; -- POP x
0c : 901; -- PUSH k
0d : 801; -- PUSHI 1
0e : 200; -- SUB
0f : 701; -- POP k
10 : f04; -- JMP L00
11 : 900; -- PUSH x
12 : b00; -- PRINT
END;

```

9.3.2 Parser Development

From the program name YACC, i.e., yet another compiler-compiler [365], we see that at the time YACC was developed it was an often performed task to write a parser for each new μ P. With the popular GNU UNIX equivalent **Bison**, we have a tool that allows us to define a grammar. Why not use **Flex** to do the job, you may ask? In a grammar we allow recursive expressions like $a + b$, $a + b + c$, $a + b + c + d$, and if we use **Flex** then for each algebraic expression it would be necessary to define the patterns and actions, which would be a large number even for a small number of operations and operands.

YACC or **Bison** both use the Bakus–Naur form or BNF that was developed to specify the language Algol 60. The grammar rules in **Bison** use terminals and nonterminals. Terminals are specified with the keyword `%token`, while nonterminals are declared through their definition. **YACC** assigns a numerical code to each token and it expects these codes to be supplied by a lexical analyzer such as **Flex**. The grammar rule use a look-ahead left recursive parsing (LALR) technique. A typical rule is written like

```
expression : NUMBER '+' NUMBER { $$ = $1 + $3; }
```

We see that an expression consisting of a number followed by the add symbol and a second number can be reduced to a single expression. The associated action is written in `{}` parenthesis. Say in this case that we add element 1 and 3 from the operand stack (element 2 is the add sign) and push back the result on the value stack. Internally the parser uses an FSM to analyze the code. As the parser reads tokens, each time it reads a token it recognizes, it pushes the token onto an internal stack and switches to the next state. This is called a *shift*. When it has found all symbols of a rule it can *reduce* the stack by applying the action to the value stack and the reduction to the parse stack. This is the reason why this type of parser is sometimes called a shift-reduce parser.

Let us now build a complete **Bison** specification around this simple add rule. To do so we first need the formal structure of the **Bison** input file, which typically has the extension `*.y`. The **Bison** file has three major parts:

```
%{
C header and declarations come here
%}
Bison definitions ...
%%
Grammar rules ...
%%
User C code ...
```

It is not an accident that this looks very similar to the **Flex** format. Both original programs **Lex** and **YACC** were developed by colleagues at AT&T [365, 367] and the two programs work nicely together as we will see later. Now we are ready to specify our first **Bison** example `add2.y`

```

/* Infix notation add two calculator */
%{
#define YYSTYPE double
#include <math.h>
void yyerror(char *);
%}

/* BISON declarations */
%token NUMBER
%left '+'

%% /* Grammar rules and actions follows */
program : /* empty */
          | program exp '\n'    { printf("%lf\n", $2); }
          ;
exp      : NUMBER           { $$ = $1; }
          | NUMBER '+' NUMBER   { $$ = $1 + $3; }
          ;
%% /* Additional C-code goes here */

#include <ctype.h>
int yylex(void)
{ int c;
  /* skip white space and tabs */
  while ((c = getchar()) == ' ' || c == '\t');
  /* process numbers */
  if (c == '.' || isdigit(c)) {
    ungetc(c,stdin);
    scanf("%lf", &yyval);
    return NUMBER;
  }
  /* Return end-of-file */
  if (c==EOF) return(0);
  /* Return single chars */
  return(c);
}

/* Called by yyparse on error */
void yyerror(char *s) { printf("%s\n", s); }

int main(void) { return yyparse(); }

```

We have added the token **NUMBER** to our rule to allow us to use a single number as a valid expression. The other addition is the **program** rule so that the parser can accept a list of statements, rather than just one statement. In the C-code section we have added a little lexical analysis that reads in operands and the operation and skips over whitespace. **Bison** calls the routine **yylex** every time it needs a token. **Bison** also requires an error routine **yyerror** that is called in case there is a parse error. The main routine for **Bison** can be short, a **return yyparse()** is all that is needed. Let us now compile and run our first **Bison** example.

```
bison -o -v add2.c add2.y
gcc -o add2.exe add2.c -lm
```

If we now start the program, we can add two floating-point numbers at a time and our program will return the sum, e.g.,

```
user: add2.exe
user: 2+3
add2: 5.000000
user: 3.4+5.7
add2: 9.100000
```

Let us now have a closer look at how **Bison** performs the parsing. Since we have turned on the **-v** option we also get an output file that has the listing of all rules, the FSM machine information, and any shift-reduce problems or ambiguities. Here is the output file **add2.output**

Grammar

```
rule 1    program ->/ * empty */
rule 2    program -> program exp '\n'
rule 3    exp -> NUMBER
rule 4    exp -> NUMBER '+' NUMBER
```

Terminals, with rules where they appear

```
$ (-1)
'\n' (10) 2
'+' (43) 4
error (256)
NUMBER (257) 3 4
```

Nonterminals, with rules where they appear

```
program (6)
  on left: 1 2, on right: 2
exp (7)
  on left: 3 4, on right: 2
```

```

state 0
$default reduce using rule 1 (program)
program go to state 1

state 1
program -> program . exp '\n' (rule 2)
$    go to state 7
NUMBER shift, and go to state 2
exp   go to state 3

state 2
exp  -> NUMBER . (rule 3)
exp  -> NUMBER . '+' NUMBER (rule 4)
'+' shift, and go to state 4
$default reduce using rule 3 (exp)

state 3
program -> program exp . '\n' (rule 2)
'\n'shift, and go to state 5

state 4
exp  -> NUMBER '+' . NUMBER (rule 4)
NUMBER shift, and go to state 6

state 5
program -> program exp '\n' . (rule 2)
$default reduce using rule 2 (program)

state 6
exp  -> NUMBER '+' NUMBER . (rule 4)
$default reduce using rule 4 (exp)

state 7
$    go to state 8

state 8
$default accept

```

At the start of the output file we see our rules are listed with separate rule values. Then the list of terminals follow. The terminal NUMBER, for instance, was assigned the token value 257. These first lines are very useful if you want to debug the input file, for instance, if you have ambiguities in your grammar rules this would be the place to check what went wrong. More

about ambiguities a little later. We can see that in normal operation of the FSM the shifts are done in states 1, 2, and 4, for the first number, the add operation, and the second number, respectively. The reduction is done in state 6, and the FSM has a total of eight states.

Our little calculator has many limitations, it can, for instance, not do any subtraction. If we try to subtract we get the following message:

```
user: add2.exe
user: 7-2
add2: parse error
```

Not only is our repertoire limited to adds, but also the number of operands is limited to two. If we try to add three operands our grammar does not yet allow it, e.g.,

```
user: 2+3+4
add2: parse error
```

As we see the basic calculator can only add two numbers not three. To have a more-useful version we add a recursive grammar rule, the operations $*$, $/$, $-$, $^$, and a symbol table that allows us to specify variables. The C-code for the symbol table can be found in examples in the literature, see, for instance, [364, p. 23], [368, p. 15], or [369, p. 65]. The lexical analysis for **Flex** is shown next.

```
/* Infix calculator with symbol table, error recovery and
   power-of */
%{
#include "ytab.h"
#include <stdlib.h>
void yyerror(char *);

%%
[a-zA-Z]      { yyval = *yytext - 'a';
                  return VARIABLE; }

[0-9]+        { yyval = atoi(yytext);
                  return INTEGER; }

[-+()^=/*\n]   { return *yytext; }

[\t] ;         /* skip whitespace */

.
yyerror("Unknown character");

%%
```

```
int yywrap(void) { return 1; }
```

We see that we now also have VARIABLES using the small single characters `a` to `z` besides the integer NUMBER tokens. `yytext` and `yylval` are the text and value associated with each token. Table 9.5 shows the variables used in the Flex↔Bison communication. The grammar for our more-advanced calculator `calc.y` now looks as follows:

```
%{
#include <stdio.h>
#include <math.h>
#define YYSTYPE int
void yyerror(char *);
int yylex(void);
int symtable[26];
%}

%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'
%left NEG      /* Negation, i.e. unary minus */
%right '^'    /* exponentiation           */

%%
program:
    program statement '\n'
    | /* NULL */
    ;

statement:
    expression
    | VARIABLE '=' expression
    ;

expression:
    INTEGER
    | VARIABLE
    | expression '+' expression
    | expression '-' expression
    | expression '*' expression
    | expression '/' expression
        if ($3) $$ = $1 / $3;
        else { $$=1; yyerror("Division by zero !\n"); }
    /* Exponentiation */
```

Table 9.5. Special functions and variables used in the **Flex**↔**Bison** communication, see Appendix A [364] for a full list

Item	Meaning
char *yytext	Token text
file *yyin	Flex input file
file *yyout	Flex file destination for ECHO
int yylength	Token length
int yylex(void)	Routine called by parser to request tokens
int yylval	Token value
int yywrap(void)	Routine called by the Flex when end of file is reached
void yyparse();	The main parser routine
void yyerror(char *s)	Called by yyparse on error

```

| expression '^' expression    { $$ = pow($1, $3); }
/* Unary minus */
| '-' expression %prec NEG      { $$ = -$2; }
| '(' expression ')'
;
%%

void yyerror(char *s) { fprintf(stderr, "%s\n", s); }

int main(void) { yyparse(); }

```

There are several new things in this grammar specification we need to discuss. The specification of the terminals using `%left` and `%right` ensures the right associativity. We prefer that $2 - 3 - 5$ is computed as $(2 - 3) - 5 = -6$ (i.e., left associative) and not as $2 - (3 - 5) = 4$, i.e., right associativity. For the exponentiation `^` we use right associativity, since 2^2^2 should be grouped as $2^(2^2)$. The operands listed later in the token list have a higher precedence. Since `*` is listed after `+` we assign a higher precedence to multiply compared with add, e.g., $2 + 3 * 5$ is computed as $2 + (3 * 5)$ rather than $(2 + 3) * 5$. If we do not specify this precedence the grammar will report many reduce-shift conflicts, since it does not know if it should reduce or shift if an item like $2 + 3 * 5$ is found.

For the divide grammar rule we have introduced error handling for divide by zero. If we had not used this kind of error handling a divide by zero would terminate the calculator, like

```

user: 10/0
calc: Floating exception (core dumped)

```

producing a large core dump file. With the error handling we get much smoother behavior, e.g.,

```
user: 30/3
calc: 10
user: 10/0
calc: Division by zero !
```

but `calc` would allow us to continue operation.

The grammar rules are now written in a recursive fashion, i.e., an expression can consist of `expression operation expression` terms. Here are a few more examples that also show the use of left and right associativity.

```
user: 2+3*5
calc: 17
user: 1-2-5
calc: -6
user: x=3*10
user: y=2*5-9
user: x+y
calc: 31
user: #
calc: Unknown character
calc: parse error
```

Any special unknown character will stop the calculator. The book CD contains the C source code as well as an executable you can play with.

Let us now briefly look at the compilation steps and the `Flex`↔`Bison` communication. Since we first need to know from `Bison` which kind of token it expects from `Flex` we run this program first, i.e.,

```
bison -y -d -o ytab.c calc.y
```

This will generate the files `ytab.c`, `ytab.output`, and `ytab.h`. The header file `ytab.h` contains the token values:

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define INTEGER 257
#define VARIABLE 258
#define NEG 259
extern YYSTYPE yylval;
```

Now we can run `Flex` to generate the lexical analyzer. With

```
flex -olexyy.c calc.l
```

we will get the file `lexyy.c`. Finally we compile both C source files and link them together in `calc.exe`

```
gcc -c ytab.c lexyy.c
gcc ytab.o lexyy.o -o calc.exe -lm
```

The **-lm** option for the math library was used since we have some more-advanced math function like **pow** in our calculator.

We should now have the knowledge to write a more-challenging task. One would be a program **c2asm** that generates assembler code from a simple C-like language. In [370] we find such code for a three-address machine. In [368] all the steps to produce assembler code for a C-like language for a stack machine are given. [360, 366, 371, 372] describe more-advanced C compiler designs. For a stack-like machine we would allow as input a file for a factorial code as

```
x=1;
scanf k;
while (k != 1) {
    x = x * k;
    k = k - 1;
}
print x;
```

The program reads data from the import (e.g., 8-pin DIP switch) calculates the factorial and outputs the data to a two-digit seven segment display, as we have on the UP2 or DE2 boards. Using the **c2asm.exe** program from the book CD, see uP, we would get

```
PUSHI    1
POP      x
SCAN
POP      k
LOO:    PUSH    k
        PUSHI   1
        CNE
        CJP     L01
        PUSH    x
        PUSH    k
        MUL
        POP      x
        PUSH    k
        PUSHI   1
        SUB
        POP      k
        JMP     L00
L01:    PUSH    x
        PRINT
```

We can then use our **asm2mif** utility (see p. 671) to produce a program that can be used to generate an MIF file that can be used in a stack machine. All

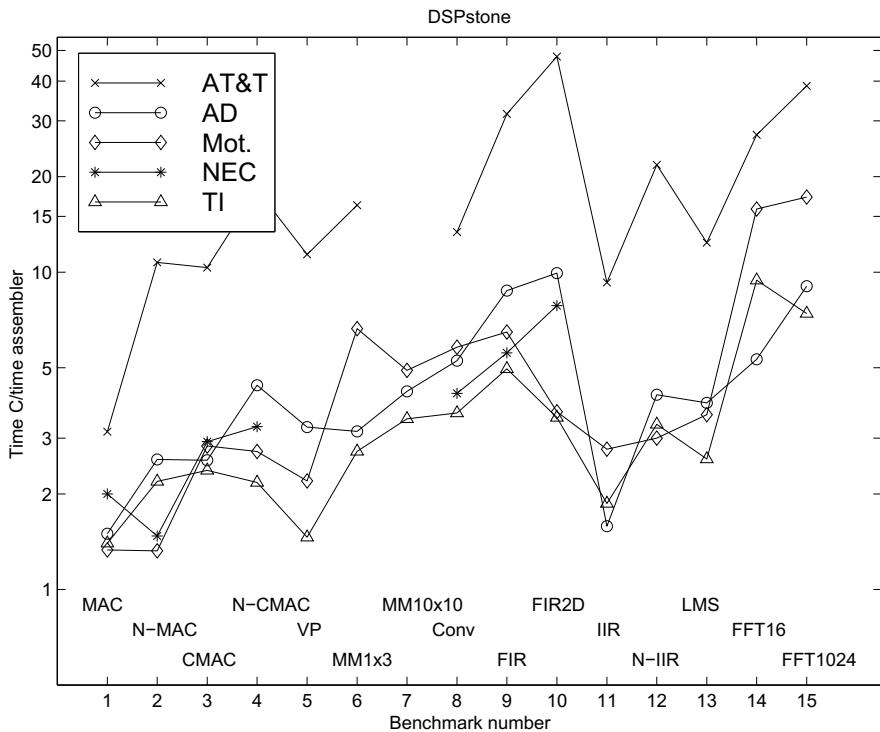


Fig. 9.14. The 15 benchmarks from the DSPstone project

that is needed is to design a stack machine, which will be discussed in the next section.

A challenge that remains is the design of a good C compiler for a PDSP due to the dedicated registers and computational units like the address generators in PDSPs. The DSPstone benchmark developed at ISS, RWTH Aachen [373, 374] uses 15 typical PDSP routines ranging from simple MACs to complex FFT to evaluate the code produced by C compilers in comparison with hand-written assembler code. As can be seen from the DSPstone benchmark shown in Fig. 9.14, the GCC-based compiler for AT&T, Motorola, and Analog PDSPs produce on average 9.58 times less-efficient code than optimized assembler code. The reason the GCC retargetable compiler was used is that the development of a high-performance compiler usually requires an effort of 20 to 50 man-years, which is prohibitively large for most projects, given the fact that PDSP vendors have many different variations of their PDSPs and each requires a different set of development tools. At the cost of less-optimized C compiler much shorter development time can be observed, if retargetable compilers like GNU GCC or LCC are used [366, 372]. A solution to this critical design problem has been provided by the associated compiler

Table 9.6. DSP additions to traditional µP [375]

Company	Part	Key DSP additions
ARM	ARM9E	Single-cycle MAC
Fujitsu	SPARClite family	Integer MAC and multimedia assistance
IBM	PowerPC family	Integer MAC
IDT	79RC4650 (MIPS)	Integer MAC
MIPS Technologies	MIPS64 5Kc	Single-cycle integer MAC
Hewlett-Packard	PA_8000 family	Registers for MPEG decode
Intel	Pentium III	Streaming SIMD extensions
Motorola	PowerPC G4	Vector processor
SUN Microsystems	UltraSPARC family	VIZ imaging instruction set

experts (ACE). ACE provides a highly flexible, easy-retargetable compiler development system that creates high-quality, high-performance compilers for a broad spectrum of PDSPs. They have implemented many optimizations for the intermediate code representation derived from the high-level language.

9.4 FPGA Microprocessor Cores

In recent years we have seen that traditional µPs have been enhanced to enable DSP operations more efficiently. In addition for some processor like Intel Pentiums or SUN SPARC we have seen the addition of MMX multimedia instruction extensions and VIZ instruction set extensions, that allows graphics and matrix multiplication more efficiently. Table 9.6 gives an overview of enhancements of traditional µPs.

The traditional RISC µP has been the basis for FPGA designs for hardcore as well as softcore processors. CISC processors are usually not used in embedded FPGA applications. Xilinx used the PowerPC as a basis for their very successful Virtex II PRO FPGA family that includes 1-4 PowerPC RISC processors. Altera decided to use the ARM processor family in their Excalibur FPGA series, which is one of the most popular RISC processors in embedded applications like mobile phones. Although these Altera FPGAs are still available they are no longer recommended for new designs. Since the performance of the processor also depends on the process technology used and that ARM-based FPGAs are not produced in the new technology, we have witnessed that Altera's softcore Nios II processor achieves about the same performance as ARM922T-based hardcore µPs. Xilinx also offer a 32-bit RISC softcore, called MicroBlaze and an 8-bit PicoBlaze processor. No Xilinx 16-bit softcore processor is available and since 16 bit is a good bit width for DSP algorithms we design such a machine in Sect. 9.5.2, p. 706. The newest addition to this set are the high performance ARM cortex-A9

Table 9.7. Dhystone µP performance

µP name	Device used	Speed (MHz)	D-MIPS measured	Hard/Soft
Nios II/e	Stratix	330	50	S
MicroBlaze	Spartan-3(-4)	85	68	S
MicroBlaze	Virtex-II PRO-7	150	125	S
V1 ColdFire	Stratix	145	135	S
ARM Cortex-M1	Stratix	200	160	S
Nios II/s	Stratix	270	170	S
ARM922T	Excalibur	200	210	H
MIPS32	Stratix	290	300	S
Nios II/f	Stratix	290	340	S
PPC405	Virtex-4 FX	450	700	H
ARM Cortex-A9	Arria/Cyclone/Zynq	800	4000	H

dual processors that are available with Altera Arria V and Cyclone V devices and Xilinx Zynq-7000 devices.

Table 9.7 gives an overview of the performance of FPGA hard- and softcore µPs measured in Dhystone MIPS (D-MIPS). D-MIPS is a collection of rather short benchmark compared with the SPEC benchmark most often used in computer architecture literature. Some of the SPEC benchmarks would probably not run, especially on softcore processors.

9.4.1 Hardcore Microprocessors

Hardcore microprocessor, although not as flexible as softcore processors, are still attractive due to their relative small die sizes and the higher possible clock rate and Dhystone MIPS rates. In the past Xilinx had favored the PowerPC series from IBM and Motorola, while Altera has used the ARM922T core, a standard core used in many embedded applications like mobile phones. Recently both vendors have introduced devices that are based on the ARM Cortex-A9 µPs. Let us have in the following a brief look at these three architectures.

Xilinx PowerPC. The Xilinx hardcore processor used in Virtex II PRO devices is a fully featured PowerPC 405 core. The RISC µP has a 32-bit Harvard architecture and consists of the following functional unit, shown in Fig. 9.15:

- Instruction and data cache, 16 KB each
- Memory management unit with 64-entry translation lookaside buffer (TLB)
- Fetch & decode unit
- Execution unit with thirty two 32-bit general-purpose registers, ALU, and MAC

- Timers
- Debug logic

The PPC405 *instruction and data caches* are both 16 KB in size while they are organized in 256 lines each with 32 bytes, i.e., $2^8 \times 32 \times 8 = 2^{16}$. Since data and program are separated this should give a similar performance as a four-way set associate cache in a von Neuman machine. The cache organization along with higher speed are the major reasons that the PPC405 outperforms a softcore, although here the cache size can be adjusted to the specific application, but are usually organized as direct mapped caches; see Exercise 9.29, p. 734.

The key feature of the 405 core can be summarized as follows:

- Embedded 450+ MHz Harvard architecture core
- Five-stage data path pipeline
- 16 KB two-way set associative instruction cache
- 16 KB two-way set associative data cache
- Hardware multiply/divide unit

The instruction cache unit (ICU) delivers one or two instructions per clock cycle over a 64-bit bus. The data cache unit (DCU) transfers 1,2,3,4, or 8 bytes per clock cycle. Another difference to most softcores is the branch prediction, which usually assumes that branches with negative displacements are taken. The execution unit (EXU) of the PPC405 is a single issue unit that contains a register file with 32 32-bit general-purpose registers (GPRs), an ALU, and a MAC unit that performs all integer instructions in hardware. As with typical RISC processors a load/store method is used, i.e., reading and writing of ALU or MAC operation is done with GPRs only. Another features usually not found in softcores is the MMU that allows the PPC405 to address a 4 GB address space. To avoid access to the page table a cache that keeps track of recently used address mappings, called the translation look-aside buffer (TLB) with 64 entries, is used. The PPC also contains three 64-bit timers: the programmable interval timer (PIT), the fixed interval timer (FIT), and a watchdog timer (WDT). All resources of the PPC405 core can be accessed through the debug logic. The ROM monitor, JTAG debugger, and instruction trace tool are supported. For more details on the PPC405 core, see [376].

The auxiliary processor unit (APU) is a key embedded processing feature that has been added to the PowerPC for Virtex-4 FX devices,⁵ see Table 1.4, p. 11. Here is a summary of the most interesting APU features [377]:

- Supports user defined instructions
- Supports up to four 32-bit word data transfers in a single instruction
- Allows to build a floating-point or general-purpose coprocessor
- Supports autonomous instructions, i.e., no pipeline stalls

⁵ This section was suggested by A. Vera from UNM.

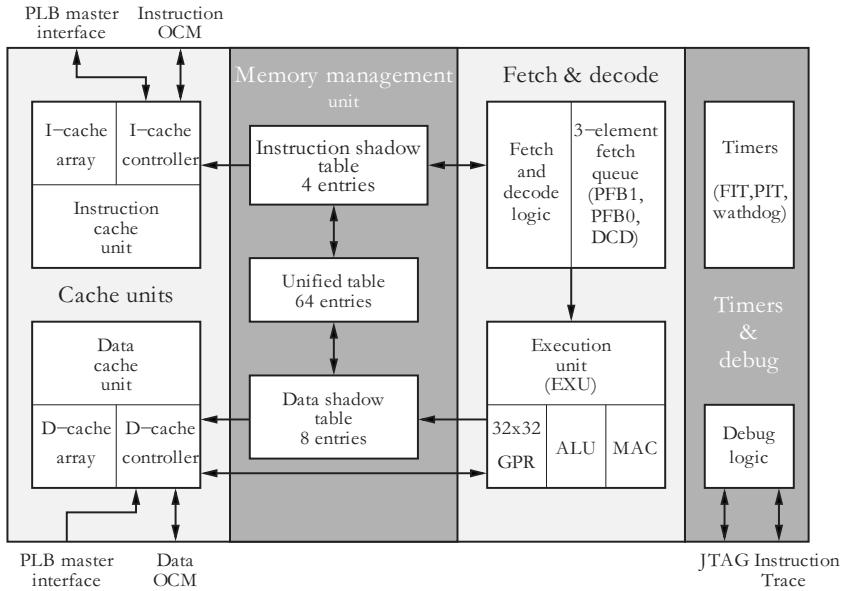


Fig. 9.15. The PPC405 core used by Xilinx Virtex II PRO devices

- 32-bit instruction width and 64-bit data
- four cycle cache line transfer

The APU is hooked up directly to the PowerPC pipeline and assembler or C-code instructions allow the access of this unit. For floating-point FIR filters improvements by a factor of 20 over software emulations have been reported [377]. A 16-bit integer 8×8 pixel 2D-IDCT block interfaced via the APU results also in a speed-up factor of about 20. By comparison, if the IDCT hardware is connected via the PowerPC local bus and not through the APU the system performance will be reduced. This is caused by the PowerPC local bus arbitration overhead and the large number of 32-bit load/store instructions required in the 8×8 pixel 2D-IDCT block [377].

Altera's ARM. Altera has included in the Excalibur FPGA family the ARM922T hardcore processor, which includes the ARM9TDMI core, instruction and data caches, a memory management unit (MMU), debug logic, an AMBA bus interface, and a coprocessor interface. The key features of the ARM922T core can be summarized as follows:

- Embedded 200 MHz (210 Dhrystone MIPS) Harvard architecture core
- Five-stage data path pipeline
- 8 KB 64-way set associative instruction cache
- 8 KB 64-way set associative data cache
- Hardware multiply unit

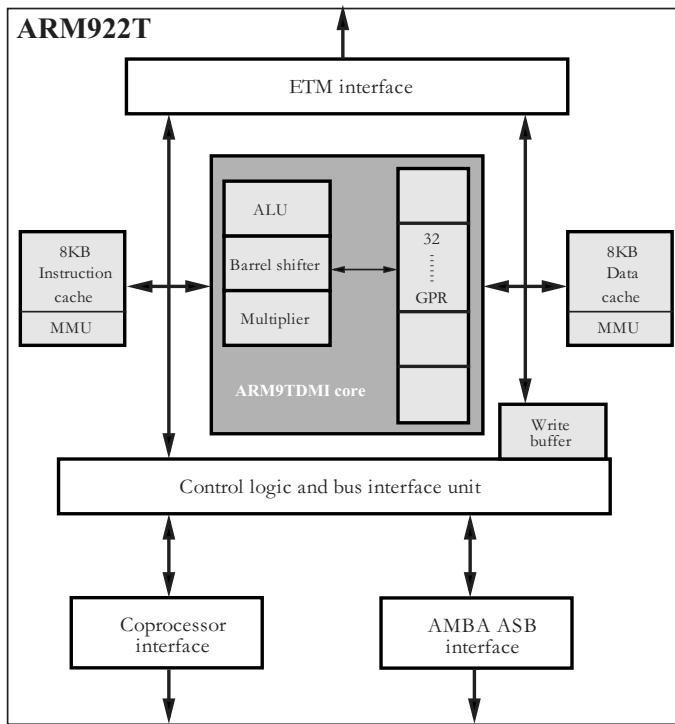


Fig. 9.16. The ARM922T overall architecture [378]. The ARM9TDMI core internal architecture is shown in dark gray [379]

- Low-power 0.8 mW/MHz; small size 6.55 mm²
- Three-operand 32-bit instructions or
- Two-operand 16-bit thumb instructions

The MMU and cache architecture used in these embedded processor are usually more sophisticated and complex and are the reason the Dhrystone MIPS rate is higher than for a softcore running with the same clock rate. The ARM922T uses an 8-word-per-line architecture. Both data and instruction cache a 64-way set-associate. In addition the ARM922T included a write buffer with 16 data words and four addresses to avoid stalling in case of cache miss. The MMU can map memory of sizes as small as 1 KB to 1 MB pages. The MMU uses two separate 64-entry translation lookaside buffers (TLBs) for the data and instructions.

The advanced microprocessor bus architecture (AMBA) is an often used bus architecture in embedded systems and is supported by the ARM922T.

The ARM9TDMI core is shown in Fig. 9.16 as the grey area. The core can operate on standard 32-bit instructions as well on the shorter thumb set that uses 16 bits only and allows one to pack two instructions in one 32-

bit memory word. The core uses a five-stage pipeline that has the following sequence: (1) fetch, (2) decode and register read, (3) execute, (4) memory access, and multiply completion, (5) write register. The CPU contains 31 general-purpose registers, with 16 registers visible at a time, while the others are reserved for context switching. Register 15 is used as the PC, register 14 holds return address for subroutine calls, and register 13 is usually used as the stack pointer. Besides the registers the core includes an ALU, a barrel shifter, and a hardware multiplier. In the following we will briefly study the thumb instruction coding.

Example 9.4: Thumb Instruction Coding

The instructions show first the regular 32-bit coding and then the thumb coding of the same instruction. The first instruction ADDS an 8-bit immediate value to a register and stores the result in the same register, i.e., $Rd = Rd + imm_{8}$

1110	00101001	Rd ₄	Rd ₄	0000	imm ₈
------	----------	-----------------	-----------------	------	------------------

The thumb instruction keeps the immediate length at 8 bits, but the two register must be the same, and the register selection is reduced from 16 to 8 registers to fit in the 16-bit instruction:

00110	Rd ₃	imm ₈
-------	-----------------	------------------

The arithmetic shift right (ASR) instruction allows one to divide a signed number by a power-of-two value. The shift amount is specified as 5-bit immediate. Rd is the destination register and Rm the source, i.e., $Rd = Rm \ggg imm_{5}$. For the standard 32-bit instruction all 16 register can be used, i.e.,

1110	00011011	SBZ ₄	Rd ₄	imm ₅	100	Rm ₄
------	----------	------------------	-----------------	------------------	-----	-----------------

while for the thumb encoding only the first eight can be used as the source and destination to meet the 16-bit instruction length, i.e.,

00010	imm ₅	Rm ₃	Rd ₃
-------	------------------	-----------------	-----------------

The multiply operation produces a 32-bit result only, and our source and destination must be the same in the thumb instruction, i.e., $Rd = (Rm * Rd)_{32}$. The thumb encoding has the following format:

010000	1101	Rm ₃	Rd ₃
--------	------	-----------------	-----------------

while the equivalent 32-bit instruction has the following format:

1110	00000001	Rd ₄	SBZ ₄	Rd ₄	1001	Rd ₄
------	----------	-----------------	------------------	-----------------	------	-----------------

9.4

As we can see from the previous example the 16-bit thumb instruction set preserves most of the feature of the 32-bit ISA, however many operations are now of the two-operand form, i.e., have to share one operand, and the number of register is reduced from 16 to 8.

Some of the more-complex instructions like the multiply-accumulate instruction MLA, which is actually a four-operand operation in the ARM922T, has no equivalent in the thumb instruction set, as the following example shows.

Example 9.5: The 32-bit instruction MLA to compute $Rd = (Rm * Rs) + Rn_{32}$ is coded as follows:

cond4	0000001	S	Rd4	Rn4	Rs4	1001	Rm4
-------	---------	---	-----	-----	-----	------	-----

It is interesting to note that it is a four-operand operation and therefore will not fit in the thumb ISA.

9.5

The fact the the MLA (a.k.a. MAC) is not included in the 16-bit thumb set is particular unfortunate if you think of DSP operations that have many MACs.

ARM Cortex-A9 on Xilinx and Altera devices. Xilinx and Altera have both introduced new device families that incorporate the ARM Cortex-A9 dual processor. Altera's Arria V and Cyclone V devices and Xilinx Zynq-7000 devices include the new A9 dual core. The Cortex-A9 versions used by both vendors are almost the same such that most likely the additional features and hard IPs included on the devices may be the key point which vendor we will choose. Xilinx devices have the dual 12-bit 1 MSPS ADC and larger onchip memory that may allow one to include the boot operating system on chip. Altera has a faster transmitter (up to 100 Gbps) and more logic resources, i.e., LE and multipliers with their families.

Now let us have a closer look at the ARM core that is shown in Fig. 9.17. It has many of the standard features we expect today from a modern 32-bit μ P, such as:

- 800 MHz dual core processor
- Dual issue superscalar pipeline with 2.5 DMIPS per MHz
- 32 KB instruction and 32 KB data L1 4-way set-associative cache
- Shared 512 KB, 8-way associate L2 cache for both processors
- 32-bit timer and watchdog

and also some additional advanced features such as

- Dynamic branch prediction
- Out-of-order multi-issue instruction queue with speculation
- Register renaming of 32 architectural to 56 physical registers
- NEON media processing accelerator for 128-bit SIMD processing
- Single and double precision floating-point operation support for $+, -, *, /$, and square-root
- Thumb-2 technology for code compression
- Configurable 32-, 64-, or 128-bit AMBA AXI interface
- Support for many I/O standards such as CAN, I²C, USB, Ethernet, SPI, and JTAG
- MMU that works with L1 and L2 to ensure coherent data

The operating system support for the ARM A9 is provided by multiple sources. There are open source tools such as Linux, Android 2.3, and FreeRTOS. In addition, commercial OS support is available such as WindRiver Linux or VxWorks, iVeia Android, or Xilinx PetaLinux.

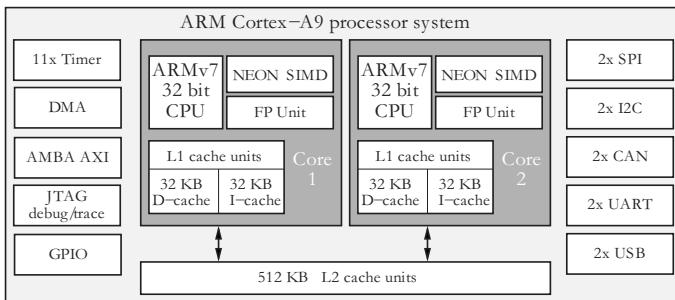


Fig. 9.17. The ARM Cortex-A9 overall architecture [380]

9.4.2 Softcore Microprocessors

Altera and Xilinx provide their own proprietary µP softcores. These processors do not try to reproduce an industry-standard processor but rather take advantage of the special hardware elements available with the FPGAs. The Xilinx PicoBlaze, for instance, makes use of the feature that the LEs can be used as dual-port RAMs, resulting in very small area requirements; Altera's Nios processor replaces the register file with M9K memory blocks that allow one to save a large number of LEs.

The most popular FPGA-based industry-standard processors are offered by third-party vendors through the FPGA vendor partner programs. We find here popular embedded µP softcores such as Motorola's 68HC11, Microchip's PIC, or the TMS320C25 from Texas Instruments. Let us now have a closer look at these FPGA-based softcore processors.

An 8-bit processor: the Xilinx PicoBlaze. Several 8-bit FPGA softcores are available for many instruction sets like Intel's 8080 or 8051, Zilog's Z80, Microchip's PIC family, MOS Technology's 6502 (popular in early Apple and Atari computer), Motorola/Freescales 68HC11, or Atmel AVR microprocessors. At www.edn.com/microprocessor a full list of current controllers is provided. The 8-bitters have become the favorite controllers. Sales are about 3 billion controllers per year, compared with 1 billion 4- or 16/32-bit controllers. The 4-bit processors usually do not have the required performance, while 16- or 32-bit controllers are usually too expensive.

One of the most important driving forces in the microcontroller market has been the automotive and home appliance areas. In cars, for instance, only a few high-performance microcontrollers are needed for audio or engine control; the other more than 50 microcontrollers are used in such functions as electric mirrors, air bags, speedometer, and door locking, to name just a few. Xilinx PicoBlaze fits right into these popular 8-bit applications and provides a nice and free-of-charge development platform. The assembler/link/loader and VHDL code for the core are available royalty free. Optimized for Xilinx devices (the low-level LUT implementation of many functions like the ALU

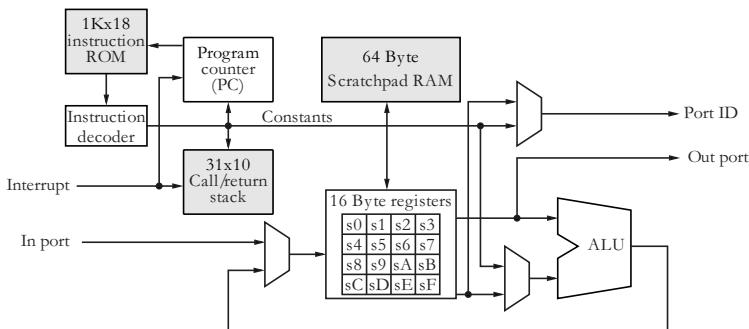


Fig. 9.18. The PicoBlaze a.k.a. KCPSM core from Xilinx

and register file using dual-port memories would make it hard to use an Altera device) the core is very small, and characterized by the following key features and performance (depending on the device family used) data; see Fig. 9.18:

- 16-byte-wide general-purpose data registers
- 256-1024 instruction words
- Byte-wide ALU operation with carry and zero flags
- 64-byte internal scratchpad RAM
- 256 input/output ports
- Four to 31 locations of CALL/RETURN stack
- Each instruction takes two clock cycles
- Twenty-one MIPS for CoolRunner II to 100 MIPS for Virtex-4
- Instruction size 16 to 18-bits
- Eight to 16 8-bit registers
- Size 76-96 slices (Virtex/Spartan) or 212 macrocells in CoolRunner-II

A free C compiler is also available written by Francesco Poderico; it is royalty free and available for download, see www.xilinx.com.

Example 9.6: The following C-code segment:

```
// DSPstone benchmark 1
char a, b, c, d;
void main()
{ d = c + a * b; }
```

will be translated into the following assembler code for the PicoBlaze:

```
;*****
; Picoblaze Small C Compiler for Xilinx PicoBlaze
; Picoblaze C Compiler for PicoBlaze, Version alpha 1.7.7
;*****

NAMEREG sf , XL
NAMEREG se , YL
NAMEREG sd , ZL
```

```

NAMEREG sc , XH
NAMEREG sa , ZH
NAMEREG sb , TMP
NAMEREG s9 , SH
NAMEREG s8 , SL
NAMEREG s7 , KH
NAMEREG s6 , KL
NAMEREG s5 , TMP2
CONSTANT _a ,ff
CONSTANT _b ,fe
CONSTANT _c ,fd
CONSTANT _d ,fc
LOAD YL , fc
JUMP _main

;// DSPstone benchmark 1
;char a, b, c, d;
;void main(){
_main:
;d = c + a*b;
INPUT ZL ,_c
SUB YL , 01
OUTPUT ZL,(YL)
INPUT ZL ,_a
SUB YL , 01
OUTPUT ZL,(YL)
INPUT ZL ,_b
INPUT XL,(YL)
ADD YL , 01
LOAD XH,XL
AND XH,80
JUMP Z,L2
LOAD XH,ff
L2:
LOAD ZH,ZL
AND ZH,80
JUMP Z,L3
LOAD ZH,ff
L3:
call _sign_mult
INPUT XL,(YL)
ADD YL , 01
ADD XL , ZL
OUTPUT XL,_d
;}_end_main: jump _end_main; end of program!

; MULT SUBROUTINE
_mult:
LOAD TMP , Of
LOAD SL , XL

```

```

LOAD SH, XH
LOAD XL, 00
LOAD XH, 00
_m1: SRO ZH
SRA ZL
JUMP NC , _m2
ADD XL , SL
ADDCY XH , SH
_m2: SLO SL
SLA SH
SUB TMP , 01
JUMP NZ , _m1
LOAD ZL,XL
LOAD ZH,XH
RETURN
_sign_mult:
LOAD TMP2,00
LOAD TMP,XH
AND TMP,80
JUMP Z,_check_member2
LOAD TMP2,01
XOR XL,ff
XOR XH,ff
ADD XL,01
ADDCY XH,00
_check_member2:
LOAD TMP,ZH
AND TMP,80
JUMP Z,_do_mult
LOAD TMP2,01
XOR ZL,ff
XOR ZH,ff
ADD ZL,01
ADDCY ZH,00
_do_mult:
CALL _mult
AND TMP2,01
JUMP NZ,_invert_mult
RETURN
_invert_mult:
XOR XL,ff
XOR XH,ff
ADD XL,01
ADDCY XH,00
RETURN

;0 error(s) in compilation

```

9.6

As can be seen from the example the C compiler uses many instructions for this short DSP code sequence; even worse because the PicoBlaze, as most

8-bitter, does not have a hardware multiply, which is done by a series of shift and adds that slows down the program further.

Although Altera does not promote its own 8-bitter, the AMPP partner supports several instruction sets, like 8081, Z80, 68HC11, PIC and 8051; see Table 9.8. For Xilinx devices we find besides the PicoBlaze also support for 8051, 68HC11, and PIC ISA.

Table 9.8. FPGA 8-bitter ISA support. Vendors: DI= Dolphin Integration (France); CI= CAST Inc.(NJ, USA); DCD=Digital core design (Poland); N/A= Information not available

μ P name	Device used	LE / Slices	BRAM/ M9Ks	Speed (MHz)	Vendor
C8081	EP1S10-5	2061	3	108	CI
CZ80CPU	EP1C6-6	3897	—	82	CI
DF6811CPU	Stratix-7	2220	4	73	DCD
DFPIC1655X	Cyclone-II-6	663	N/A	91	DCD
DR8051	Cyclone-II-6	2250	N/A	93	DCD
Flip8051	Xc2VP4-7	1034	N/A	62	DI
DP8051	Spartan-III-5	1100	N/A	73	DCD
DF6811CPU	Spartan-III-5	1312	N/A	73	DCD
DFPIC1655X	Spartan-III-5	386	3	52	DCD
PicoBlaze	Spartan-III	96	1	88	Xilinx

Notice how the low-level hardware optimization of the PicoBlaze with only 177 four-input LUTs makes it the smallest and fastest 8-bitter for Xilinx devices.

An 16-bit processor: the Altera Nios. The first generation Nios embedded processor is a configurable RISC processor with a 16- or 32-bit datapath. Nios embedded systems can be created with any number of peripherals. Figure 9.19 shows the SOPC builder 32-bit standard configuration of the Nios processor.

Table 9.9 shows the base core sizes for the Nios embedded processor and some of the IP core peripherals that integrate with the standard Nios embedded processor to form complete microprocessing units. Most peripherals can be parameterized to fit the specific application and can be instantiated multiple times within a single μ P. In addition, customer-designed logic and peripherals can be integrated with the Nios processor to deliver a unique μ P. The creation of these custom μ Ps can be done in minutes using the Altera SOPC builder tool, and synthesized to run on any Altera FPGA. In addition to the IP cores listed in Fig. 9.19, SOPC builder features additional IP cores available from Altera and Altera's megafunction partners program (AMPP).

Nios processors lower than version 2.0 have a three-stage pipeline (load, decode, execute) and each instruction takes a predictable amount of time.

Table 9.9. Nios core and peripheral sizes, logic elements (LE) count, and embedded array blocks (M9K)

Unit	LE	M9K
16-bit data path Nios	950	2
32-bit datapath Nios	1250	3
UART, fixed baud rate	170	
Timer	244	
Serial peripheral interface(SPI):		
8-bit master, one slave	103	
SPI: 8-bit master, two slaves	108	
General-purpose I/O: 32-bit, tristate	138	
SDRAM controller	380	
External memory/peripheral: 32-bit	110	
External memory/peripheral: 16-bit	85	

For versions later than 2.0 the three-stage pipeline is replaced by a five-stage pipeline with sophisticated prefetch logic, interlocking, and hazard management. The pipeline logic hides these details from the programmer and makes it more difficult to analyze the execution time just via instruction count only, since the latency of the instruction depends on many factors, like the pre- or post instruction, operands, and memory location, to name just a few. Altera provide a best-case estimate for each instruction the actual latency

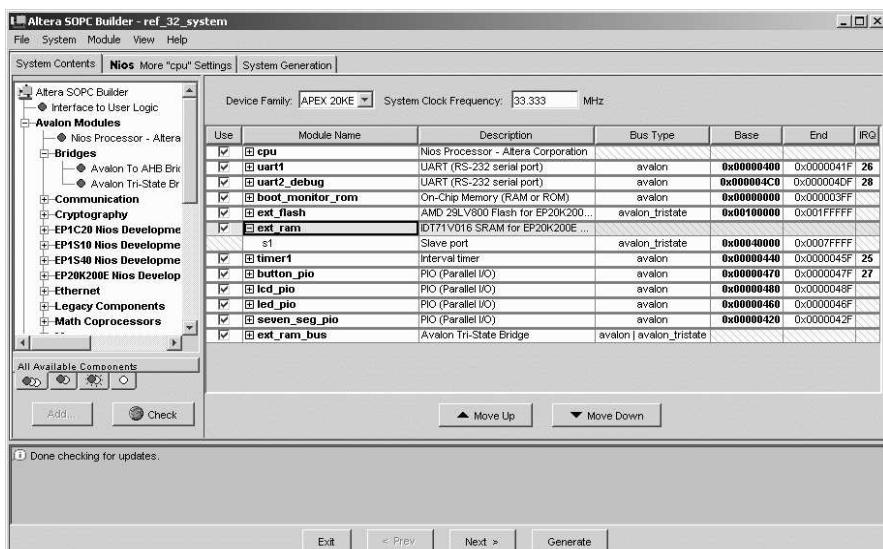


Fig. 9.19. SOPC Nios 32-bit standard processor template

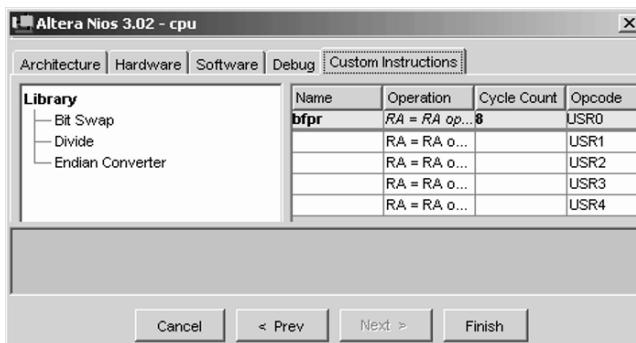


Fig. 9.20. Custom instruction features of the Nios processor

Table 9.10. Altera clock cycles in 5 stage pipeline Nios processors [381]

Function name	memory location	Clock cycle	Comment
ASR, ASRI, LSL, LSLI, LSR, LSRI	—	1	Shift operations
MUL	—	2	$16 \times 16 \rightarrow 32\text{-bit}$
JMP, CALL	—	2	control flow
LD, ST	on-chip	2	Load and store
TRET	—	3	Return function
TRAP	—	4	Hold processor
LD, ST	off-chip	4	Load and store

however maybe longer. The minimum clock-cycle estimate for some typical instructions is shown in Table 9.10.

The Altera Nios differs from other softcore processor solutions in the market by including custom instruction features; see Fig. 9.20. Custom instruction design is a process of implementing a complex sequence of standard instructions in hardware in order to reduce them to a single-instruction macro that can be accessed by software. The custom instructions can be used to implement complex processing tasks in single-cycle (combinatorial) and multi-cycle (sequential) operations. In addition, these user-added custom instructions can access memory as well as logic outside the Nios system. As an example design in the case study section, we will study a custom implementation of the bitreverse operation used in radix-2 FFT and DCT.

With the wide range of densities available in FPGA devices and the small sizes of Nios embedded systems, system designers can divide complex problems into smaller tasks and use multiple Nios embedded processors. These Nios processors can be customized with a wide selection of peripherals, defining very simple to very complex μ P systems. By targeting low-cost devices, powerful, customized embedded systems can be realized at the best cost in the industry.

Table 9.11. Nios processor core multiplier options

Multiplication option	Clock cycles 32-bit product	Hardware effort
Software	80	0
MSTEP	18	14-24 LEs
MUL	3	427-462 LEs

The Nios processor shown in Fig. 9.21 has a pipelined general-purpose RISC architecture [353, 382–385]. The 32-bit processor has a separate 16-bit instruction bus and 32-bit data bus. The register file is configurable to have 128, 256 or 512 registers but at one time only 32 of these registers are accessible as general-purpose registers through software using a sliding window. These large numbers of internal registers are used to accelerate subroutine calls and local variable access. The CPU template is in general configurable with instruction and data cache memory, which increases its performance. The Nios instruction set can be configured to increase software performance and can be modified either by adding custom instructions or by using pre-defined instruction set extensions provided with the processor template. The three predefined multiplier optimizations for the Nios processor are listed in Table 9.11, giving the number of clock cycles and size required for each of the multiplier options:

- The MUL instruction includes a hardware 16×16 -bit integer multiplier.
- The MSTEP instruction provides the hardware to execute one step of a 16×16 -bit multiply in one clock cycle.
- Software multiplication uses the C runtime libraries to implement integer multiplication with sequences of shift and add instructions.

Any of these three multiplier options could be used to implement 16×16 -bit multiplication in software. Depending on the overall processor architecture the additional hardware effort may vary.

Alternative 16/24-bit microprocessors are offered by IP vendors, that re-build standard PDSPs (Motorola 56000; TI TMS320C25) or GPP (Motorola 68000). Table 9.12 give an overview of available core and required resources.

An 32-bit processor: the Xilinx MicroBlaze. As an example of a 32-bit softcore processor let us in the following study the Xilinx MicroBlaze. The MicroBlaze is a Harvard 32-bit data and instruction RISC processor, that is available with three or five pipeline stages. The standard key features of the MicroBlaze core can be summarized as follows:

- MicroBlaze area optimized is a three-pipeline-stage core with 1.03 DMIPS per MHz
- MicroBlaze performance optimized is a five-pipeline-stage core with branch optimizations delivering 1.38 DMIPS/MHz

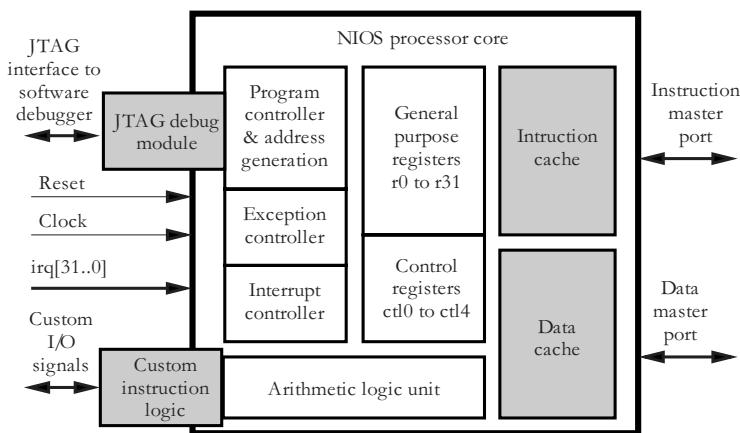


Fig. 9.21. Nios processor core. (Grey blocks are optional)

Table 9.12. FPGA 16/24-bit ISA support. Vendors: CI= CAST Inc.(NJ, USA); DCD=Digital core design (Poland)

μ P name	Device used	LE / slices	BRAM	Speed (MHz)	Vendor
C32025TX	Stratix II	3916	18 M4K	68	CI
C68000	Stratix V	4429	—	114	CI
D68000	Cyclone-6	6604	n/a	44	DCD
C80186EC	Stratix IV	8042 LEs	—	90 MHz	CI
C32025 PDSP	Kintex-7	983 slices	—	159	CI
D68000	Virtex-II PRO-7	3415	n/a	65	DCD

- The ALU, shifter, and 32×32 register file are standard

Optional items that can be included at configuration time are:

- Barrel shifter
- Array multiplier
- Divider
- Single precision floating-point unit for add, subtraction, multiply, divide, and comparison
- Data cache from 2–64 KB
- Instruction cache from 2 to 64 KB

The five-stage pipeline is executed in the following steps: (1) fetch, (2) decode, (3) execute, (4) memory access, and (5) writeback.

The data and instruction caches have a direct mapped cache architecture. The cache can be accessed in blocks of four or eight words. One or more Block-

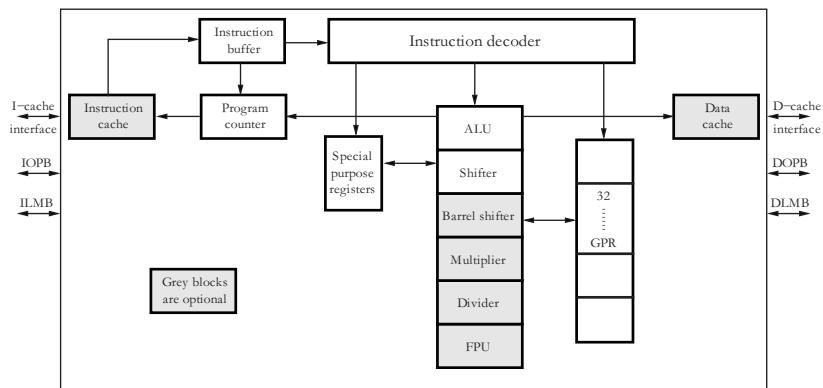


Fig. 9.22. The MicroBlaze softcore architecture core from Xilinx

RAMs are used to store the data, while an additional BlockRAM is used to store the tag data. Let us study a typical cache configuration.

Example 9.7: MicroBlaze Cache Configuration

Let us design in the following a cache that uses only two BlockRAMs. Since BlockRAMs in Spartan-3 and Spartan-6 are of size 16 Kbits, we conclude, for 32-bit word size, that we can store 512×32 words in a single BlockRAM. Using a cache size of less than 16 Kbits or 2 KB will not really save any resources and that is why this is the smallest available cache size. Now we have to decide if we want to use the four or eight words-per-line configuration. Usually with eight words per line we can address a larger external memory, while four words per line gives a faster decoder, so let us start with the four word per line and see what the maximum external memory we can address is.

With 512 words in our cache, and four words per line we conclude that we have to store 128 tags in our tag memory. Then our tag BlockRAM needs to be configured as 128×32 memory. Each line now needs a valid bit and four valid bits for each word, which leaves up to 27 bits for the tags. Therefore the external memory will be limited to an address space of 27 tag bits plus 11 LSBs used to address 2 KB by the cache, i.e., a 38-bit word can be addressed, i.e., $2^{38} = 256$ GB memory. This is probably much larger than the actual main memory and more than the 32-bit address space of the MicroBlaze. This configuration (with 13 tag bits to address the 16 MB of the Nexsys board) is shown in Fig. 9.23.

At the upper end Xilinx allows one to use a 64 KB cache memory. We then need 32 BlockRAMs for the cache data alone. If we now use one more BlockRAM to store the tags, we again have to decide if we use the four or eight words-per-line configuration. The 64 KB requires 16 K words, and 4 K lines for the four words-per-line configuration. With 4 K lines we then need to use the $2^{12} \times 4$ BlockRAM configuration. We need four valid and one line bit, so one BlockRAM for the tags will not be enough. With two BlockRAMs for the tags we will have three bits for tags, i.e., the main memory can have a size of 512 KB. With each additional BlockRAM we can then increase the main memory by a factor of 16.

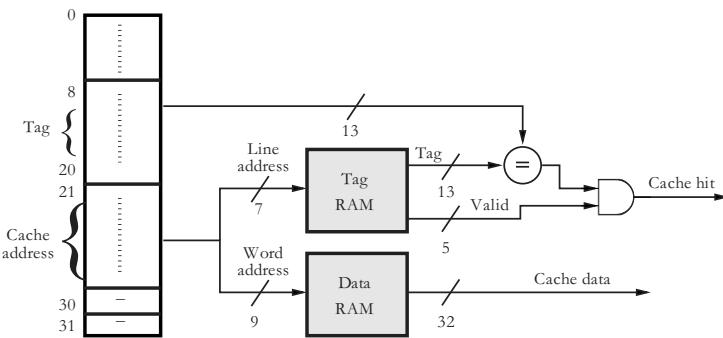


Fig. 9.23. The MicroBlaze 2KB cache configuration for 16 MB main memory on the Nexys boards

Table 9.13. DMIPS comparison for different memory organizations (D=data; I=program memory)

External SRAM		Internal Cache		BRAM		LEs	DMIPS
D	I	D	I	D	I		
✓	✓	—	—	—	—	8718	7.139
✓	✓	✓	✓	—	—	9076	29.13
—	✓	—	—	✓	—	8812	47.81
—	—	—	—	✓	✓	8718	59.78

Other example configurations are discussed in Exercises 9.33 and 9.34, p. 735.

Given the option that we can now use a cache memory for the MicroBlaze one question is still open: what is the best memory configuration for maximum performance? This question has been evaluated by Fletcher [386] and the results are shown in Table 9.13. For almost all embedded microprocessor, when the FPGA just hosts the processor and the program and data are kept in external SRAM memory, the performance improves if we add an on-chip data and/or program cache, see rows 4 and 5 in Table 9.13. However, if we are able to keep the whole main memory inside the FPGA this will be better than a design with external memory plus caches, as Table 9.13 shows for the Dhrystone benchmarks, which are much smaller than the SPEC benchmarks in a conventional computer and fit inside the FPGA BlockRAMs.

Alternative 32-bit microprocessors are offered by Altera and IP vendors, which rebuild custom or standard GPP (e.g., Motorola 68000). The Altera Nios II is available in three different versions: A fast six-stage pipeline version that provides 1.13 DMIPS/MHz; a standard version with five pipeline stages

that delivers 0.64 DMIPS/MHz; and an economy version with minimal core size and a one-stage pipeline that provides 0.15 DMIPS/MHz. Table 9.14 gives an overview of available core and required resources.

Table 9.14. FPGA 32-bit ISA support. Vendors: CI= CAST Inc.(NJ, USA); N/A = data not available

μ P name	Device used	Area	BRAM/ M9Ks	Speed	Vendor
C68000-AHB	Stratix II	4053 ALUT	5	98 MHz	CI
Nios-II fast	Stratix IV	900 ALM	N/A	340 DMIPS	Altera
Nios-II std	Stratix V	700 ALM	N/A	170 DMIPS	Altera
Nios-II eco.	Stratix V	350 ALM	N/A	50 DMIPS	Altera
C68000-AHB	Virtex-6	1466 slices	5	125 MHz	CI

9.5 Case Studies

Finally let us have a look at three more-detailed design projects. The first is the HDL design of a complete zero-address, i.e., stack machine, that uses the assembler and C compiler tools we developed in Sect. 9.3, p. 660. The second case study is a LISA-based DWT processor design that shows the wide variety (simple μ P to true vector processor) that can be built with a few LISA instructions [387]. The final case study shows how a custom DSP block can be tightly couple with Altera's Nios processor. The bitreverse addressing of a FFT processor is chosen and hardware optimizations are discussed [388,389].

9.5.1 T-RISC Stack Microprocessors

Let us start our design explorations with a simple stack machine. Although the stack machine is called a zero-address machine, we still need a couple of bits in the instruction to define immediate operands. If we select 8-bit data and 4-bit instructions then we can define 16 instructions and have a 12-bit data word that is easily represented in the simulation by three half bytes. We can choose seven ALU instructions (0-6), five data move instructions (7-11), and four control flow instructions. More specifically, our instructions set then becomes:

- The ALU instructions use the top of the stack (TOS) and the second of the stack (if applicable) and can be further split into:
 - Four arithmetic operations: ADD, SUB, MUL, and INV
 - Three logic operations: OPAND, OPOR, and OPNOT

- Data move instructions move data from/to the top of the stack:
 - POP <var> stores the TOS data word memory location **var**.
 - PUSH <var> loads a data word **var** from memory and places it on TOS.
 - PUSHI <imm> puts the immediate value **imm** on TOS.
 - SCAN reads 8 bits from the input port and puts them on TOS.
 - PRINT writes 8 bits from the TOS to the output port.
- The four program control instructions are:
 - CNE and CEQ compare the two top elements of the stack and set the jump control register JC accordingly.
 - CJP <imm> loads the PC with the value **imm** if the JC is set to true.
 - JMP <imm> loads the PC with the value **imm**.

Since with Cyclone IV devices we can only implement synchronous memory, we have to use a register for the input data or address. This makes it impossible to have a PC update, program, data, and TOS update in one clock cycle and a minimum of two clock cycles must be used. Figure 9.24a shows the implemented timing. The PC is update on the first falling edge. This update is used as the input for the address of the program memory. The output data of the program memory is stored by the next rising edge in the PROM output register. The instructions are then decoded, and only for the POP operation do we store TOS with the next falling edge in the data memory. For any ALU or PUSH operation the memory is routed through the ALU and stored in the TOS register with the next rising edge. At the same time we update the values in the stack. A four-value stack as used in the classic HP41 pocket calculators should provide enough stack depth. Since such a short stack is used, it is much easier to use registers than a LIFO M9K memory block. The proposed timing works for all instructions except the control flow instructions. Since the comparison values has to be stored in the JC register we need an extra clock cycle to implement a conditional jump operation. In the first step we update the JC register and in the next clock cycle the PC is updated according to the JC register; see Fig. 9.24b.

Example 9.8: A Stack Machine

The following VHDL code⁶ shows the four-entry stack machine. As a test program, factorial computation (C-code see p. 680; MIF code see p. 671) is shown in the simulation.

```
-- Title: T-RISC stack machine 4/e
-- Description: This is the top control path/FSM of the
-- T-RISC, with a single 3 phase clock cycle design
-- It has a stack machine/0-address type instruction word
-- The stack has only 4 words.
LIBRARY ieee; USE ieee.std_logic_1164.ALL;

PACKAGE n_bit_int IS                      -- User defined types
  SUBTYPE SLVA IS STD_LOGIC_VECTOR(7 DOWNTO 0);
```

⁶ The equivalent Verilog code **trisco.v** for this example can be found in Appendix A on page 875. Synthesis results are shown in Appendix B on page 881.

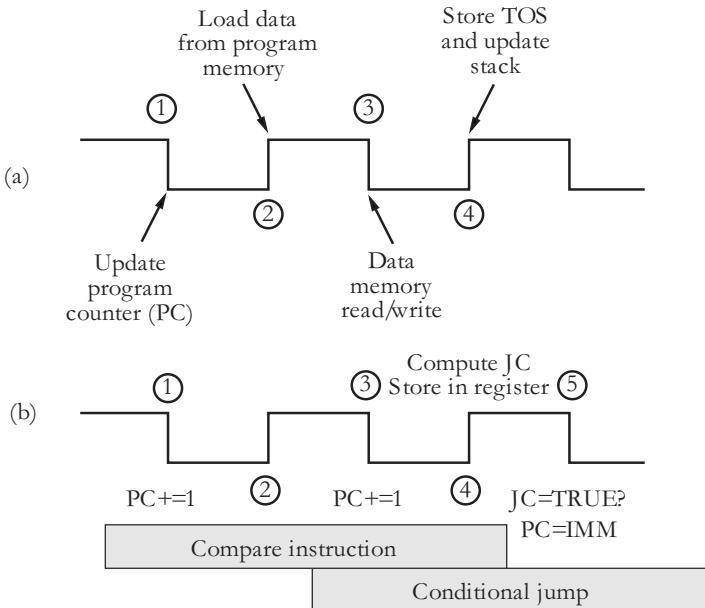


Fig. 9.24. Timing for T-RISC operations. (a) Usual four clock edges used in most instructions. (b) Two-instruction sequence timing used for conditional jump instructions

```

SUBTYPE SLVD IS STD_LOGIC_VECTOR(7 DOWNTO 0);
SUBTYPE SLVP IS STD_LOGIC_VECTOR(11 DOWNTO 0);
END n_bit_int;

LIBRARY work;
USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.STD_LOGIC_1164.ALL;
USE ieee.STD_LOGIC_arith.ALL;
USE ieee.STD_LOGIC_signed.ALL;

ENTITY trisc0 IS
  GENERIC (WA : INTEGER := 7; -- Address bit width -1
           WD : INTEGER := 7); -- Data bit width -1
  PORT(clk      : IN STD_LOGIC; -- System clock
        reset    : IN STD_LOGIC; -- Asynchronous reset
        jc_OUT   : OUT BOOLEAN;  -- Jump condition flag
        me_ena   : OUT STD_LOGIC; -- Memory enable
        iport    : IN SLVD;      -- Input port
        oport    : OUT SLVD;     -- Output port
        s0_OUT   : OUT SLVD;     -- Stack register 0
        s1_OUT   : OUT SLVD;     -- Stack register 1
  );
END ENTITY trisc0;
  
```

```

dmd_IN : OUT SLVD;          -- Data memory data read
dmd_OUT : OUT SLVD;         -- Data memory data write
pc_OUT  : OUT SLVA;         -- Progamm counter
dma_OUT : OUT SLVA;         -- Data memory address write
dma_IN  : OUT SLVA;         -- Data memory address read
ir_imm  : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
                      -- Immidiate value
op_code : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
                      -- Operation code
END;

ARCHITECTURE fpga OF trisco IS

SIGNAL op   : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL imm, s0, s1, s2, s3, dmd : SLVD;
SIGNAL pc, dma : SLVA;
SIGNAL pmd, ir  : SLVP;
SIGNAL eq, ne, mem_ena, not_clk : STD_LOGIC;
SIGNAL jc      : boolean;

-- OP Code of instructions:
CONSTANT add   : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"0";
CONSTANT neg   : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"1";
CONSTANT sub   : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"2";
CONSTANT opand : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"3";
CONSTANT opor  : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"4";
CONSTANT inv   : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"5";
CONSTANT mul   : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"6";
CONSTANT pop   : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"7";
CONSTANT pushi : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"8";
CONSTANT push  : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"9";
CONSTANT scan  : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"A";
CONSTANT print : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"B";
CONSTANT cne   : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"C";
CONSTANT ceq   : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"D";
CONSTANT cjp   : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"E";
CONSTANT jmp   : STD_LOGIC_VECTOR(3 DOWNTO 0) := X"F";

-- Programm ROM definition and value
TYPE MEMP IS ARRAY (0 TO 19) OF SLVP;
CONSTANT prom : MEMP :=
(X"801", X"700", X"a00", X"701", X"901", X"801", X"c00",
X"e11", X"900", X"901", X"600", X"700", X"901", X"801",
X"200", X"701", X"f04", X"900", X"b00", X"FOO");

-- Data memory definition
TYPE MEMD IS ARRAY(0 TO 2**((WA+1)-1)) OF SLVD;
SIGNAL dram : MEMD;

BEGIN

P1: PROCESS (clk, reset, op) -- FSM of processor
BEGIN -- store in register ?
    CASE op IS -- always store except Branch

```

```

        WHEN pop      => mem_ena <= '1';
        WHEN OTHERS => mem_ena <= '0';
    END CASE;
    IF reset = '1' THEN
        pc <= (OTHERS => '0');
    ELSIF falling_edge(clk) THEN
        IF ((op=cjp) AND NOT jc ) OR (op=jmp) THEN
            pc <= imm;
        ELSE
            pc <= pc + "00000001";
        END IF;
    END IF;
    IF reset = '1' THEN
        jc <= false;
    ELSIF rising_edge(clk) THEN
        jc <= (op=ceq AND s0=s1) OR (op=cne AND s0/=s1);
    END IF;
END PROCESS p1;

-- Mapping of the instruction, i.e., decode instruction
op   <= ir(11 DOWNTO 8);    -- Operation code
dma  <= ir(7 DOWNTO 0);    -- Data memory address
imm  <= ir(7 DOWNTO 0);    -- Immidiate operand

prog_rom: PROCESS (clk, reset)
BEGIN
    IF reset = '1' THEN          -- Asynchronous clear
        pmd <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
        pmd <= prom(CONV_INTEGER(pc)); -- Read from ROM
    END IF;
END PROCESS;
ir <= pmd;

data_ram: PROCESS (clk, reset, dram, dma)
VARIABLE idma : INTEGER RANGE 0 TO 255;
BEGIN
    idma := CONV_INTEGER('0' & dma); -- force unsigned
    IF reset = '1' THEN          -- Asynchronous clear
        dmd <= (OTHERS => '0');
    ELSIF falling_edge(clk) THEN
        IF mem_ena = '1' THEN
            dram(idma) <= s0; -- Write to RAM
        END IF;
        dmd <= dram(idma); -- Read from RAM
    END IF;
END PROCESS;

P3: PROCESS (clk, reset, op)
VARIABLE temp: STD_LOGIC_VECTOR(2*WD+1 DOWNTO 0);
BEGIN
    IF reset = '1' THEN          -- Asynchronous clear
        s0 <= (OTHERS => '0'); s1 <= (OTHERS => '0');

```

```

      s2 <= (OTHERS => '0'); s3 <= (OTHERS => '0');
      oport <= (OTHERS => '0');
ELSIF rising_edge(clk) THEN
  CASE op IS
    -- Specify the stack operations
    WHEN pushi | push | scan => s3<=s2; s2<=s1; s1<=s0;
    -- Push type
    WHEN cjp | jmp | inv | neg => NULL;
    -- Do nothing for branch
    WHEN OTHERS => s1<=s2; s2<=s3; s3<=(OTHERS=>'0');
    -- Pop all others
  END CASE;
  CASE op IS
    WHEN add   => s0  <= s0 + s1;
    WHEN neg   => s0  <= -s0;
    WHEN sub   => s0  <= s1 - s0;
    WHEN opand  => s0  <= s0 AND s1;
    WHEN orpor  => s0  <= s0 OR s1;
    WHEN inv    => s0  <= NOT s0;
    WHEN mul    => temp := s0 * s1;
                    s0  <= temp(WD DOWNTO 0);
    WHEN pop    => s0  <= s1;
    WHEN push   => s0  <= dmd;
    WHEN pushi  => s0  <= imm;
    WHEN scan   => s0 <= iport;
    WHEN print  => oport <= s0; s0<=s1;
    WHEN OTHERS => s0 <= (OTHERS => '0');
  END CASE;
END IF;
END PROCESS P3;

-- Extra test pins:
dmd_OUT <= dmd; dma_OUT <= dma; -- Data memory I/O
dma_IN <= dma; dmd_IN <= s0;
pc_OUT <= pc; ir_imm <= imm; op_code <= op;
                                         -- Program
jc_OUT <= jc; me_ena <= mem_ena; -- Control signals
s0_OUT <= s0; s1_OUT <= s1;      -- Two top stack elements

END fpga;

```

We see in the coding first the generic definition followed by the entity that includes the ports and the test pins. The architecture part starts with general-purpose signals and then the op-code for all 16 instructions are listed as constant values. The first process in the architecture body FSM hosts the finite state machine that is used to control the microprocessor. Then program and data memory are instantiated via ROM and RAM blocks. All operations that include an update of the stack are included in ALU. The previous constant definitions allow very intuitive coding of the actions. Finally some extra test pins are assigned that are visible as output ports. The design uses 171 LEs, one M9K (VHDL coding) or two M9Ks (Verilog coding), one embedded multiplier and with a registered performance of $F_{max}=92.66$ MHz using the TimeQuest slow 85C model.

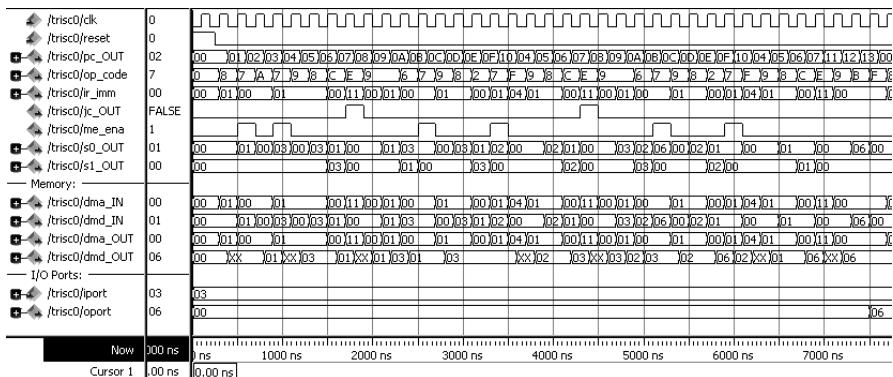


Fig. 9.25. T-RISC simulation of factorial example

The simulation from the previous design for a factorial program is shown in Fig. 9.25. The program starts by loading the input value from the `iport`. Then the computation of the factorial starts. First the loop variable is evaluated; if it is larger than 1, then `x` is multiplied by `k`. Then `k` is decremented and the program jumps to the start of the loop. After two runs through the loop the program is finished and the factorial result ($2 \times 3 = 6$) is transported to `oport`.

9.5.2 LISA Wavelet Processor Design

A microprocessor is a much more-efficient way of using FPGA resources than a direct hardware implementation of an algorithm and has become in recent years one of the most important IP blocks for FPGA vendors. Altera, for instance, reported that they sold 10,000 systems of the Nios microprocessor development systems in the first three years alone. Xilinx reported an even larger number of downloads of their MicroBlaze microprocessors.

A new generation of design tools is empowering software developers to take their algorithmic expressions straight into FPGA hardware without having to learn traditional hardware design techniques. These tools and associated design methodologies are classified collectively as electronic system-level (ESL) design, broadly referring to system design and verification methodologies that begin at a higher level of abstraction than the current mainstream HDL. The language for instruction set architecture (LISA), for instance, allows us to specify a processor instruction or cycle accurately using a few LISA operations, then explore architecture using a tool generator and profiler (see Fig. 9.26), and finally determine speed/size/power parameters via automatically synthesized VHDL or Verilog code. ESL tools have been around for a while, and many perceive that these tools are predominantly focused on ASIC design flows. But with ASIC mask charges of \$4 million in 65 nm technology the number of designs using FPGAs is rapidly increasing. In reality, an

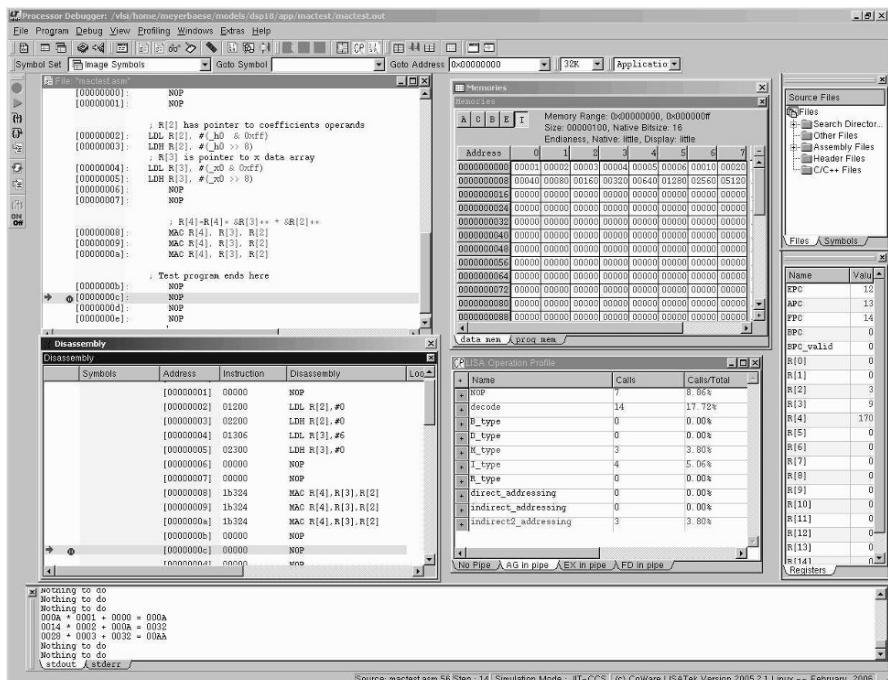


Fig. 9.26. LISA development tools: (left) disassembler, (center) memory monitor and pipeline profiles, (right) file and register window

increasing number of ESL tool providers (e.g., Celoxica, Codetronix, Synopsys, Binachip, Impulse Accelerated, Mimosys) are focusing on programmable logic.

Today the majority of microprocessors are employed in embedded systems. This number is not surprising because a typical home today may have a Laptop/PCs with a high-performance microprocessor but probably dozens of embedded systems, including electronic entertainment, household, and telecom devices, each of them equipped with one or more embedded processors. A modern car typically has more than 50 microprocessors. Embedded processors are often developed by relatively small teams with short time-to-market requirements, and the processor design automation is clearly a very important issue. Once a model of a new processor is available, existing hardware synthesis tools enable the path to FPGA implementation. However embedded processor design typically begins at a much higher abstraction level, even far beyond an instruction set and involves several architecture exploration cycles before the optimum hardware/software partitioning has been found. It turns out that this requires a number of tools for software development and profiling. These are normally written manually – a major source of cost and inefficiency in embedded processor design so far. The LISA processor design

Table 9.15. LISA example models (CC=C compiler generated)

Name	CC	Pipeline stages	Description
QSIP_X	–	3	<ul style="list-style-type: none"> • Harvard RISC architecture • 12 different tutorial versions • Single cycle ALU • Pipeline and zero pipeline version
LT_DSP_32p3	✓	3	<ul style="list-style-type: none"> • Single cycle ALU with MAC • Zero overhead loops • 32-bit instruction • 24-bit data path • 48-bit accumulator
LT_VLIW_p4	–	4	<ul style="list-style-type: none"> • QSIP like ISA • Parallel load/store • Parallel arithmetic instructions

platform (LPDP), originally developed at RWTH Aachen and now a product of Synopsys Inc. addresses these issues in a highly innovative and satisfactory manner; see Fig. 9.26. The LISA language supports profiling-based stepwise refinement of processor models down to cycle accuracy and even VHDL or Verilog RTL synthesis models. In an elegant way, it avoids model inconsistencies otherwise inevitable in traditional design flows. Microprocessors from simple RISC to highly complex VLIW processor have been described and successfully implemented using LPDP for FPGAs and cell-based ASICs.

Synopsys provides 14 different models. This include seven tutorial models that are used as part of Synopsys training material. Some have multiple versions like the more than ten different designs in the QSIP_X models. Four starting point models are provided and used as a skeleton for starting a new architecture. Three different IP models for classic architectures are also included. All models are instruction accurate and most of the models are Harvard-type RISC models that are also cycle accurate. Pipeline stages vary from three to five. Provided are all types of modern processor from simple RISC (QSIP), over PDSP like LT_DSP_32p3 to VLIW LT_VLIW_32p4 to special processors like a 16- to 4096-point FFT processor LT_FFT_48p3. Table 9.15 shows the properties of some example models.

LISA 18-bit instruction word RISC processor. Xilinx offers a 32-bit MicroBlaze and a 8-bit PicoBlaze RISC processor but no processor with 16 or 24-bit, as typically for DSP algorithms used, is offered. Let us in the following design such a 16-bit RISC machine with the LPDP. Since a 16-bit processor fits in between the Micro- and PicoBlaze we will call our RISC processor NanoBlaze.

For an FPGA design we can start with the three-pipeline RISC tutorial design of the LISA 2.0 QSIP_12 model and extend the ISA to make it more

useful for the FPGA design. The BlockRAMs in Xilinx FPGAs are 18 bits wide and the instruction words should therefore also be 18 bits wide. When using BlockRAM there will be no gain when using instruction words short than 18 bits. The byte-wide access in QSIP model should be changed to flat 18-bit instruction and data access. Changes then have to be included in the instruction counter, memory configuration *.cmd file, `step_cyle`, and the data memory instructions LDL, LDH, and LDR. The following listing show the supported instructions of the designed NanoBlaze:

- Arithmetic/logic unit (ALU) instructions:
 - ADD: three-operand add operation with two source operands and a third destination operand.
 - MUL: three-operand multiply operation with two source operands and a third destination operand. Only the lower 16 bits of the product are preserved.
- Data move instructions:
 - LDL: load the lower 8 bits of the data word with a constant value.
 - LDH: load the upper 8 bits of the data word with a constant value.
 - LDR: load register from memory. The memory location can be specified explicitly as a constant or indirectly via a general-purpose register.
 - STR: store register content to memory. The memory location can be specified explicitly or indirectly via a general-purpose register.
- Program control instructions:
 - BC: the condition branch checks a (loop) register for zero and not zero.
 - B: an unconditional branch.
 - BDS: the delay branch is a condition BC with the feature that the next instruction after the BDS instruction is also executed.

The basic instruction set of the DWT RISC processor consists of nine instructions that were designed using 28 LISA operations. The instruction coding of the instruction in the execution pipeline stage is shown in Fig. 9.27.

The NanoBlaze processor can now be synthesized and implemented in an FPGA. Depending on the type of memory used (i.e., CLB- or BlockRAM-based) we get the synthesis results shown in Table 9.16.

Example 9.9: If we now use the RISC processor to implement a length-8 DWT processor as shown in Fig. 5.57, p. 380, we need two length-8 filter $g[n]$ and $h[n]$ and for each output sample pair 16 multiply and 14 add operations are necessary. For 100 samples with an output downsampling by 2 the arithmetic requirements for the DWT filter band would therefore be $8 \times 100 = 800$ multiplications and $7 \times 100 = 700$ additions. From the Calls shown in the instruction profile in Fig. 9.28 we see that the number of multiplication is in fact 800, however the number of add instructions was more than four times higher as expected. This is due to the fact that the register updates for the memory access are also computed with the general-purpose ALU.

**Fig. 9.27.** NanoBlaze instruction set architecture**Table 9.16.** NanoBlaze synthesis result for the Xilinx device XC3S1000-4ft256

Parameter	NanoBlaze with CLB-based RAM	NanoBlaze with BlockRAM
Slices	1896	1893
4-input LUT	3443	3602
Multiplier	1	1
BlockRAMs	0	2
Total gates	32,986	162,471
Clock period	13.293 ns	13.538 ns
F_{\max}	75.2 MHz	73.9 MHz

In addition to the large number of add operations to update the memory register pointer also 1 600 LDR load operations were performed. This can be substantially improved if we use auto-increment, indirect memory access as used for the MAC operation in PDSPs.

LISA programmable digital signal processor. From the DWT processor discussed in the last section we have seen the large required arithmetic

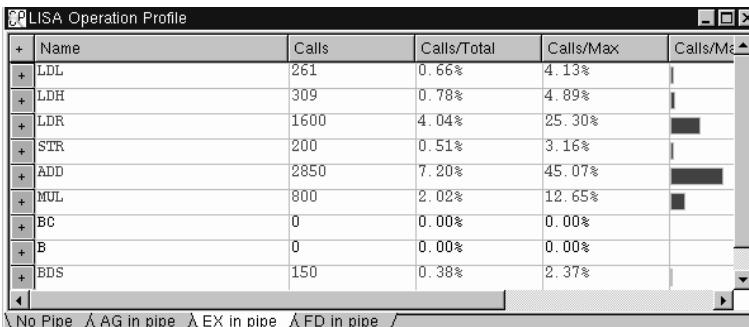


Fig. 9.28. NanoBlaze operation profile for 100-point length-8 dual-channel DWT including memory initialization, i.e., 300 instructions

count for updating the memory pointer and the memory access itself. A single multiply accumulate instruction in NanoBlaze requires the following operations:

```
; use pointer R[2] and R[3] to load operands
LDR R[8], R[2]
LDR R[9], R[3]
; increment register pointer using R[1]=1
; multiply and add result in R[4] and avoid data hazards
ADD R[2], R[2], R[1]
MUL R[7], R[8], R[9]
ADD R[3], R[3], R[1]
ADD R[4], R[4], R[7]
```

DSP algorithms typically operate on linear data arrays (i.e., vectors) and post-auto-increments or decrements in the memory pointer are therefore frequently used. In addition a fused add and multiply, usually called MAC, allows the previous six instructions to be combined into one single instruction, i.e.,

```
; load and multiply the values from pointer R[2] and R[3],
; and add the product to register R[4]
MAC R[4], R[3], R[2]
```

The ISA additions to the NanoBlaze is shown in Fig. 9.29. The addition of such a MAC operation to the instruction set requires two major modifications. First we need to provide a LISA operation that allows two indirect memory accesses. In hardware this results in a more-complex address generation unit and a dual-output-port data memory that supports two reads in one clock cycle. Secondly, we need to add the LISA operation for the MAC instructions.

The LISA operation to implement the MAC instruction can be implemented as follows:

```
/* This LISA operation implements the instruction MAC. */
```

M_type	insn	address				reg		
⊕ reg	insn	address				reg		
⊖ address	insn	address				reg		
⊖ indirect2_addressing	insn	1	reg16	aux16		reg		
⊕ aux16	insn			index		reg		
⊕ reg16	insn		index			reg		
⊖ indirect_addressing	insn	1	reg16	x	x	reg		
⊕ reg16	insn		index	x	x	reg		
⊖ direct_addressing	insn	0	imm8_addr			reg		
⊕ imm8_addr	insn	0	value			reg		
⊖ Insn	insn	address				reg		
MAC	0 1 1 0 1	address				reg		
LDR	0 1 1 1 0	address				reg		
STR	0 1 1 1 1	address				reg		

Fig. 9.29. Programmable digital signal processor (DSP18) instruction set additions

```
/* It accumulates the product of two register and stores */
/* the result in a destination register. */ */
OPERATION MAC IN pipe.EX
{
    DECLARE
    {
        REFERENCE address;
        REFERENCE reg;
    }
    CODING { 0b01101 }
    SYNTAX { "MAC" }
    BEHAVIOR
    {
        short tmp1, tmp2, s1, s2; /* Temporary */
        short tmp_reg;
        short res;

        tmp_reg = reg;

        s1 = (data_mem[EX.IN.ar] & (char)0xffff);
        s2 = (data_mem[EX.IN.ar1] & (char)0xffff);
        res = tmp_reg + s1 * s2;
#pragma analyze(off)
        printf ("%04X * %04X + %04X = %04X\n",
               s1, s2, tmp_reg, res);
#pragma analyze(on)

        reg = res;

    }
}
```

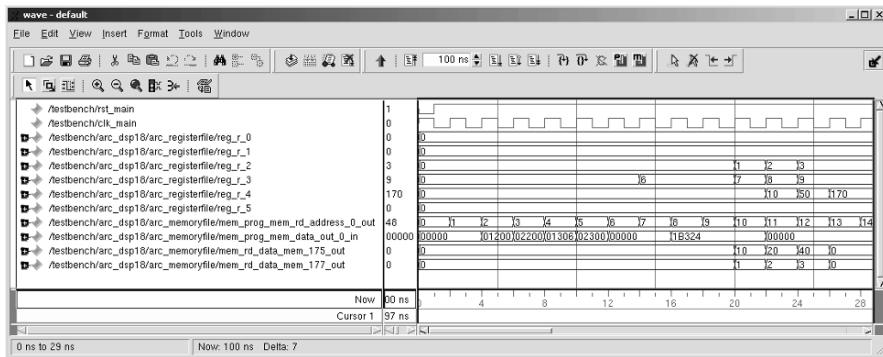


Fig. 9.30. DSP18 test bench for MAC operation

The MAC LISA operation starts with the **DECLARE** section that refers to elements that are defined in other LISA operations. The **CODING** section that describes the OP code follows. The assembler syntax would be **MAC**, and finally in the **BEHAVIOR** section the implementation of the instruction is shown. We have used an additional **printf** inside the operation to monitor progress. While this does not change the hardware, we can monitor the output of our MAC instruction directly in the debugger window; see the lower window in Fig. 9.26.

We can then go ahead and synthesize the new processor that we want to call DSP18 due to the PDSP-like added features in the instruction set and perform a test bench simulation in the MODELSIM simulator.

Example 9.10: To verify the functionality of the generated VHDL code we use the MODELSIM simulator. LPDP generated all the required HDL code (VHDL or Verilog) and all the required simulation scripts (i.e., MODELSIM *.do-files). As the test values we use $x = [1, 2, 3]; g = [10, 20, 40]$; and the MAC operation should progress as follows:

1. $\text{MAC} = 1*10=10$
2. $\text{MAC} = 2*20=40 \Rightarrow 40+10=50$
3. $\text{MAC} = 3*40=120 \Rightarrow 120+50=170$

The correct function can be seen from the MODELSIM simulation from **reg_r_4**, shown in Fig. 9.30, which shows the contents of register $r[4]$.

9.10

If we now write the program for the same 100-point length-8 DWT as in the last section, we will see the large impact the MAC operation has on the overall instruction count. Now the 800 MAC operations are the dominate operations and much fewer explicit add or memory operations are required. The total instruction count improves from 5,870 for the NanoBlaze to 1,968 for the DSP18. The operation profile (including memory initialization, i.e., 300 operations) for the DWT example is shown in Fig. 9.31. Since the DSP18 is larger and the addressing modes are more sophisticated than in the NanoBlaze, the

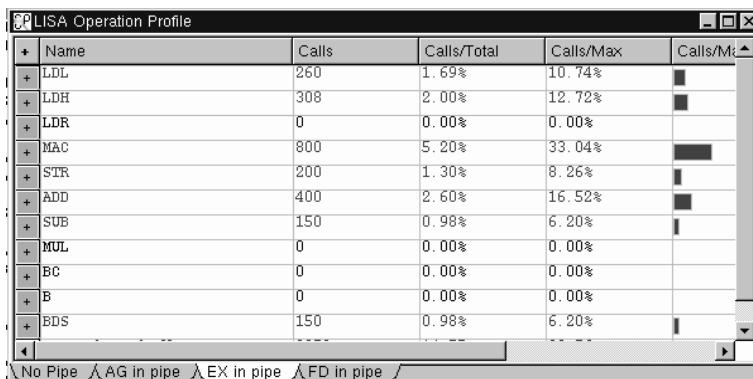


Fig. 9.31. DSP18 operation profile for 100-points length-8 dual-channel DWT including memory initialization, i.e., 300 instructions

overall registered performance decreases to 39 MHz when using CLB-based RAM and 51 MHz when using BlockRAM. Table 9.17 shows the implementation results for two different external memory configurations.

LISA true vector processor. General-purpose CPUs have improved in recent years by exploring instruction-level parallelism (ILP), adding on-chip cache and floating-point units, speculative branch execution, and improved speed etc. One particular problem that occurs now is that the logic to track dependencies between all in-flight instructions grows quadratically with the number of instructions [338]. As a result these improvements have considerably slowed down since 2002 and the use of multiple CPUs on the same die is now favored instead of increasing clock speed. This requires code to be written for parallel processors, which may be less efficient than using a vector processor to start with. Vector processors were successfully commercialized long before ILP machines and use an alternative approach to controlling multiple function units with deep pipelines. Vector processors like Cray, NEC, or Fujitsu VP100 provide high-level instructions that work on vectors, i.e., a linear array of numbers. Usually vector processor are characterized by using:

- a vector array with dedicated load/store unit
- functional unit that are highly pipelined
- hazard control is minimized
- support of vector instruction, that replace a complete loop by a single instruction

Figure 9.32 shows an experimental vector processor that is a vector extension of the popular MIPS machine called VMIPS. VMIPS was introduced in 2001 has eight vector registers each with 64 elements, one load/store, and five arithmetic units, one lane, and runs at 500 MHz.

The second DSPstone benchmark: $d[k] = a[k] \times b[k]; 0 \leq k \leq N$, for instance, would be implemented in VMIPS by

Table 9.17. DSP18 synthesis result for the XC3S1000-4ft256 Spartan-3 Xilinx device

Parameter	DSP18 with CLB-based RAM	DSP18 with BlockRAM
Slices	3145	2679
4-input LUT	6053	5183
Multiplier	2	2
BlockRAMs	0	2
Total gates	81,509	177,203
Clock period	25.542 ns	19.565 ns
F_{\max}	39.15 MHz	51.11 MHz

MULV. D V1,V2,V3

i.e., multiply the elements of the vectors V2 and V3 and put each result in the vector register V1.

However, as can be seen from Fig. 9.32, the typically implemented vector processor architecture only looks to a programmer as a vector machine. Inside the vector processor we may typically find eight vector registers where each vector has 32 to 1024 (Fujitsu VP100) elements but usually only one floating-point arithmetic unit for each operation. A vector multiply or add, like in DSPstone2, still requires N clock cycles (not counting the initialization). Multiple lanes that allow more than one floating-point operation per clock cycle are limited. In the last 30 years of vector processor history only two machines (the NEC SX/5 from 1998 and the Fujitsu VPP5000 introduced in 1999) have had over 10 lanes, but the quotient of register elements to lanes is still only 3% for the 512-elements-per-vector NEC SX/5 with 16 lanes. The reason that typically VPs do not have more than one lane is that the floating-point units in 64 bits need a large die area.

Another weakness of current vector processors is their limited usefulness for DSP operations. In DSPs we not only need a vector multiply, instead more often we need an inner product computation, i.e.,

$$\mathbf{X} \times \mathbf{Y} = \sum_{k=0}^{N-1} X[k] \times Y[k]. \quad (9.7)$$

While the multiplication can be done in a vector element-by-element parallel fashion the summation requires the addition of all products in an adder tree. This is usually not supported with vector instructions. A third operation that is not supported in most vector processors is the (cyclic) shift of the vector register elements. For instance, if an FIR application requires the vector elements $x[0] \dots x[N-1]$, then in the next step the elements $x[1] \dots x[N]$ are needed. A PDSP uses cyclic addressing to address this issue. In a vector processor it is usually necessary to reload the complete vector.

In an FPGA design we can therefore improve the processing by

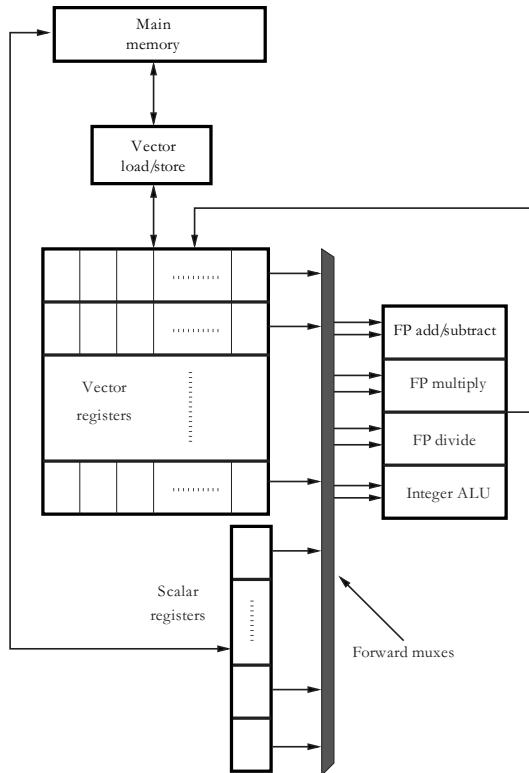


Fig. 9.32. The VMIPS vector processor

- Adding the vector shift instructions **VSXY** and **VSXY** to our instruction set to load two words from data memory, shift the two vector registers of the data or coefficients, and place the two new values in the first location.
- Since modern FPGA can have up to 512 embedded multipliers, we can implement as many multipliers as vector elements are in a vector. A **VMUL** instruction will perform 2×8 multiplications and place the products in the two product vector registers P and Q.
- Implement (inner product) vector sum instructions **VAP** and **VAQ** that adds up all elements in a (product) register vector.

We call such a machine a true vector processor (TVP) since vector operations like vector multiply are no longer translated into a sequence of single multiplies – all operations are done in parallel. For a two-channel length-8 wavelet processor we would therefore require 16 embedded multipliers. The Spartan-3 device XC3S1000-4ft256 that is used in the low-cost Nexys Dig-



Fig. 9.33. The true vector processor (TVP) instruction set additions

lent university boards, for instance, has 24 embedded 18×18 -bit multipliers available, more than enough for our TVP.

The inner product sum may be a concern in terms of speed since here horizontal $L - 1$ additions need to be performed for a vector register with L elements. But we can perform the additions on a binary adder tree, as the following LISA code examples shows for the VAP instruction:

```
/* Vector scalar add of all P register */
OPERATION VAP IN pipe.EX {
DECLARE
{
    REFERENCE dst;
}
CODING { 0b100101 }
SYNTAX { "VAP" }
BEHAVIOR
{short t1,t2,t3,t4,t5,t6,t7;
```

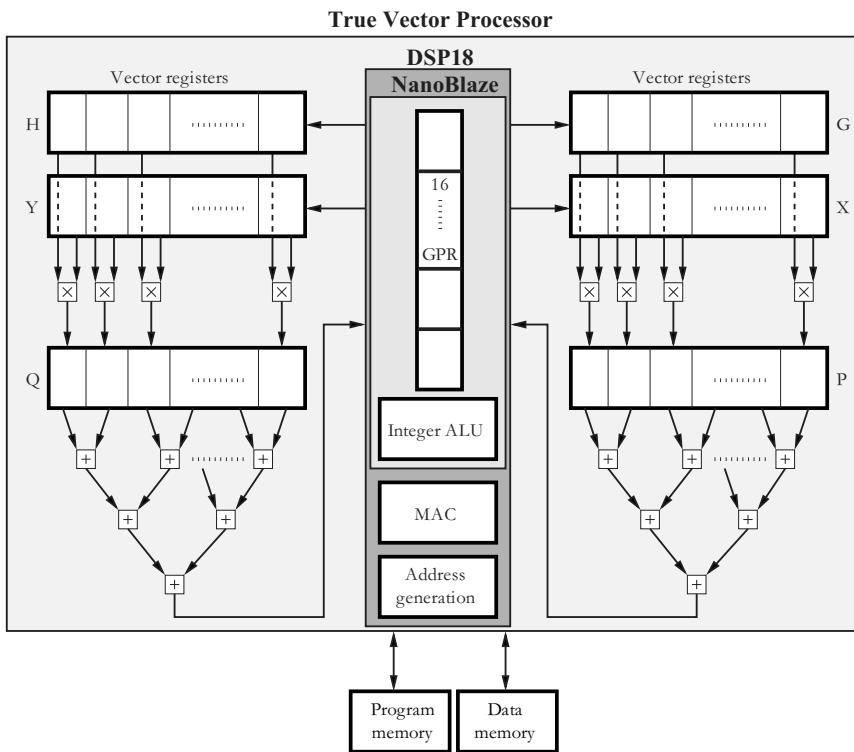


Fig. 9.34. The true vector processor (TVP) architecture

```

t1 = P[0] + P[1];
t2 = P[2] + P[3];
t3 = P[4] + P[5];
t4 = P[6] + P[7];
t5 = t1 + t2;
t6 = t3 + t4;
t7 = t5 + t6;
dst = t7;
}
    
```

which reduces the worst-case delay from seven adds to three.

If we now implement the DWT length-8 processor with the TVP ISA we find that the inner loop is much shorter, i.e., only nine instructions. The downsampling by 2 of the DWT requires two shifts of the vectors **X** and **Y** for each new output sample.

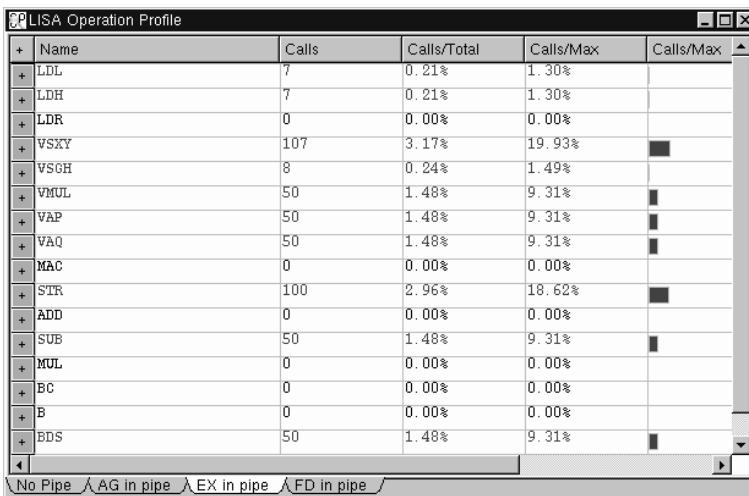


Fig. 9.35. TVP operation profile for length-8 dual-channel DWT

_loop:

```

VSXY R[2],R[3]
VSXY R[2],R[3]
VMUL
VAP R[4]
VAQ R[5]
STR R[4], R[6]
STR R[5], R[6]
BDS @_loop, R[7]
; next instruction is in the branch delay slot
SUB R[7],R[7],R[1]

```

The overall instruction count when compared with the DSP18 is further decreased, as can be seen from the profile shown in Fig. 9.35. The total instruction count for TVP is only 479 instructions.

From Table 9.18 we notice the larger resource requirements and lower maximum operation frequency.

Final LISA processor design comparison. Let us finally compare all three designs in terms of size, speed, and overall throughput in MSPS for a DWT length-8 example. The key synthesis properties are summarized in Table 9.19. The device used is a Spartan-3 XC3S1000-4ft256 as in the Nexys Digilent university boards (see <http://www.digilentinc.com/>), with 7680 slices, 15360 four-input LUTs, 24 embedded multiplier, and 24 BlockRAMs with 18 Kbit each. When data or program memory are implemented with CLB-based RAM (called distributed RAM by Xilinx) then about 800 four-

Table 9.18. TVP synthesis result for the Xilinx device XC3S1000-4ft256

Parameter	μ P only	with BlockRAM
Slices	4907	4993
4-input LUT	8850	9226
Multiplier	18	18
BlockRAMs	0	2
Total gates	141,158	274,463
Clock Period	22.082 ns	20.799 ns
F_{\max}	45.3 MHz	48.1 MHz

Table 9.19. BlockRAM synthesis results of three different DWT length-8 processor designs using LISA for Xilinx device: XC3S1000-4ft256

Parameter	NanoBlaze	DSP18	TVP
LISA operations	28	32	40
Prog. memory	$2^7 \times 18$	$2^7 \times 18$	$2^7 \times 18$
Data memory	$2^8 \times 16$	$2^8 \times 16$	$2^8 \times 16$
BRAMs	2	2	2
Gates	162,471	177,203	274,463
MHz	73.9	51.11	48.1

input LUTs are required for a $2^8 \times 16$ and about 120 four-input LUTs for a $2^7 \times 18$ memory.

The overall performance of the three processors is measured by the mega samples per second (MSPS) throughput when implementing a length-8 DWT as shown in Fig. 5.57, p. 380. We need two length-8 filter $g[n]$ and $h[n]$ and for each output sample pair 16 multiply and 14 add operations are computed. For 100 samples with an output downsampling by 2 the arithmetic requirements for the DWT filter band would therefore be $8 \times 100 = 800$ multiplications and $7 \times 100 = 700$ additions, or 800 MAC calls. From the NanoBlaze instruction profile however we see that many addition cycles for LDR and ADD are required, due to the fact that the register updates for memory access are also computed with the general-purpose ALU. The DSP18 processor reduces the LDR and ADD essentially, and although the maximum clock frequency is decreased, the overall throughput is improved by a factor of 2. If we use a true vector processor, then an inner product can be computed in two clock cycles and the overall throughput is improved by a factor of 8 compared with NanoBlaze and a factor of 4 when compared with a single-core MAC DSP18 design.

Finally, the performance data of the three LISA based processors and a direct RNS polyphase implementation (4217 LEs, 155 MSPS, [390]) are compared in Fig. 9.36. We see the large improvement from NanoBlaze to TVP, but still a direct mapping into hardware is another magnitude faster

Table 9.20. Comparison of three different DWT length-8 processor designs using LISA for the Xilinx Spartan-3 device XC3S1000-4ft256

Parameter	NanoBlaze	DSP18	TVP
LDL	259	259	7
LDH	208	208	7
LDR	1600	0	0
VSXY	—	—	107
VSGH	—	—	8
VMUL	—	—	50
VAP	—	—	50
VAQ	—	—	50
MAC	—	800	0
STR	100	100	100
ADD	2650	300	0
SUB	—	50	50
MUL	800	0	0
BC	0	0	0
BC	0	0	0
BDS	50	50	50
Total	5667	1767	479
Clock	76.28	54.98	44.72
MSPS	1.35	3.11	9.54

than any microprocessor solution. The hardware architecture, however, can only implement one configuration, while the TVP software architecture can implement many different algorithms.

9.5.3 Nios Custom Instruction Design

As you may recall from Chap. 6, in FFT and DCT algorithms the input or output values appear in bitreverse order; see Fig. 9.37. If you write down the index in binary form then all bit locations need to be switched, e.g., we go from $110 \rightarrow 011$. The memory locations 6 and 3 need to swap the values. This bitreverse operation should be the task for this CI case study.

As a starting point for our CI design we may use the Nios II design examples from TERASIC or the “Computer” systems provided in the Altera University program. The Altera University program computers are available for many of the latest Quartus II versions and several TERASIC boards such as DE0, DE1, DE2, DE2-70, and DE2-115 are supported. They come in two flavors. A simple machine called Basic Computer that includes all switches, LEDs, and SRAM and DRAM memory, and a second, bigger system, that also includes several audio and image processing IP blocks called Media Computer. For our CI case study the Basic Computer system has more than enough features and we use the Basic Computer as our starting

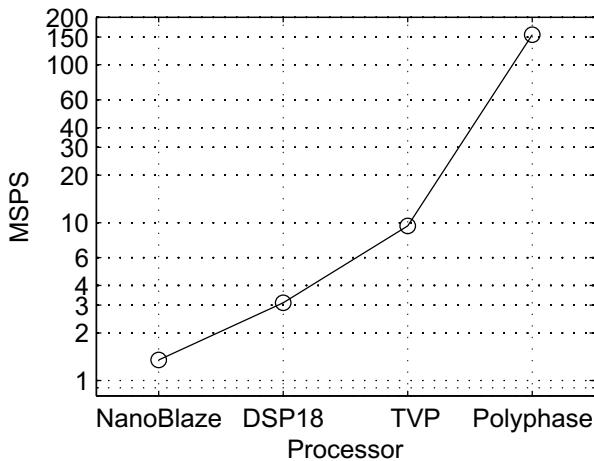


Fig. 9.36. Comparison of LISA-based processor and direct RNS polyphase implementation

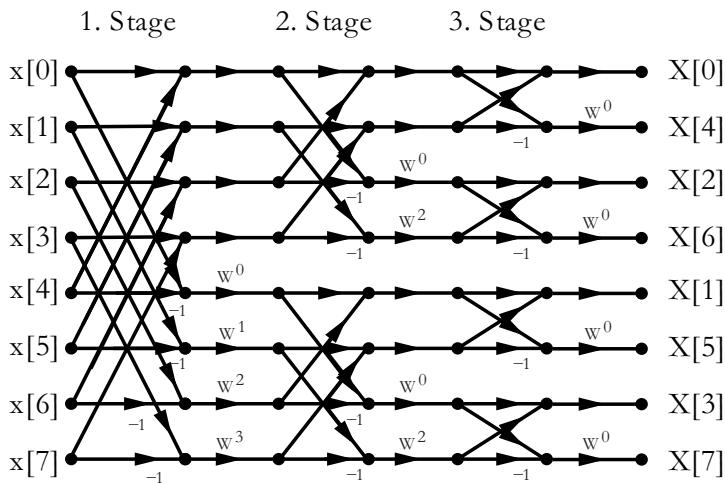


Fig. 9.37. Signal flow graph length-8 DIF FFT. The output data $X[k]$ appear in bitreverse order

point since then compile time is reduced. In general it is often easier to use a version of the Media or Basic Computer that matches your design tools and board and add the extra custom instruction (CI) files than to try to modify another system such that all cores, pin descriptions, and library files work.

Modern softcore processors allow a close integration of custom instructions (CI) and associated logic within the core without the long delay through

external bus operations. There are however a few consideration related to the processor and algorithms which should be taken into account:

- The most improvement through a CI is expected for a “slow” processor such as the Xilinx PicoBlaze or the Nios II/e. For a Nios II/f the improvement may be less significant. A highly pipelined processor may actually be run slower with a CI.
- Large improvements can be expected for algorithms only if an HW implementation is fast and compact. Bitwise operations like a bitreverse needed in DCT or FFT or switch boxes in cryptographic algorithms are good examples that should result in large improvements with a CI. On the other hand a 256-point FFT was improved by only 45-77% when using a butterfly processor [34]. The large design effort in a CI may then not justified for such a small improvement obtained for custom arithmetic circuits.

There are different types of CI in the Nios **SOPC Builder** and **Qsys** environment. Since only **Qsys** will be supported in future Quartus versions we will focus our attention on the **Qsys** environment. The basic ports in use are shown in Fig. 9.38b. The Nios II allows five types of CI:

- 1) The pure combination circuit function, that has three ports and no clock.
- 2) The fixed multi-cycle that adds a **clock** input and is required to complete the computation after a specified number of cycles.
- 3) The variable multi-cycle that uses an additional **done** port to indicate the completion of the operation.
- 4) The extended CI that has an 8-bit port **n** that allows one to multiplex different output ports.
- 5) The internal register file type CI that has 32 words for each of the 3 I/O ports.

Creation and integration of custom logic block. First we rename the Altera Basic Computer to our new project name **DE2_115_CI_Computer**. You need to change the name and contents for the ***.qsf**, ***.vhd**, and ***.qpf** files. You may also need to copy the component directory called **nios_system** to your project. The HDL is placed with the other component files under **nios_system→synthesis→submodules** by **Qsys** and can be modified later. All these CIs can easily be inferred from the **Qsys** template if you start **Qsys** and then click **New...** and select the **Template** menu. Since we like to implement a bitreverse function we use the **Combinational** template. Under **Files** we then add our component description in Verilog or VHDL, that is in our case a 4-, 8-, 12-, and 16-bit bitreverse operation. We may have to modify the port names as requested by **Qsys**. Such a circuit will be small, runs fast, and the HDL is as shown below:

```
-- VHDL Custom Instruction Template File for
-- Internal Register Logic
```

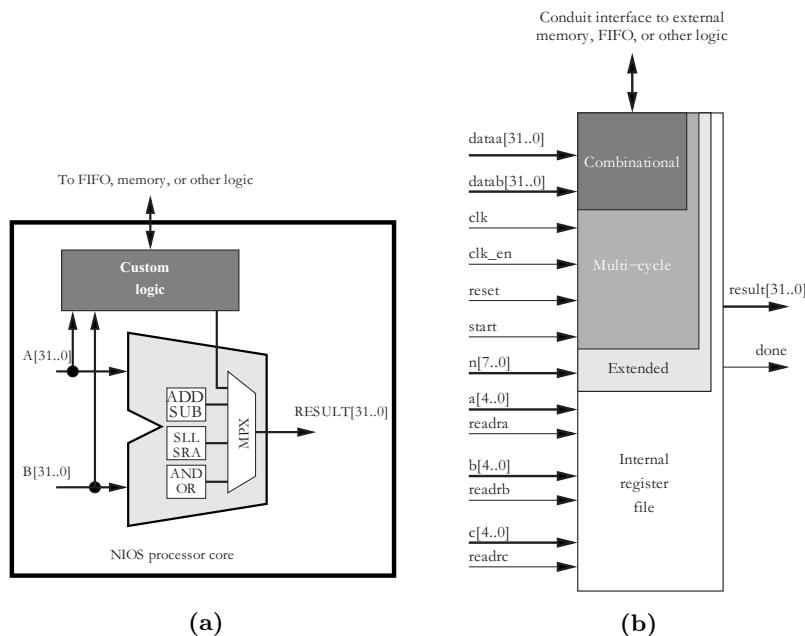


Fig. 9.38. (a) Adding custom logic to the Nios ALU. (b) Physical ports for the custom logic block

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;
-- -----
ENTITY nios_system_CI_SWAP_0 IS
PORT(
    SIGNAL clk : IN std_logic;
    SIGNAL ncs_cis0_dataaa: IN STD_LOGIC_VECTOR(31 DOWNTO 0);
                                -- Operand A (always required)
    SIGNAL ncs_cis0_datab: IN STD_LOGIC_VECTOR(31 DOWNTO 0);
                                -- Operand B (optional)
    SIGNAL ncs_cis0_result: OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
);                                -- result (always required)
END ENTITY;
-- -----
ARCHITECTURE a_custominstruction OF nios_system_CI_SWAP_0 IS

    SIGNAL b : INTEGER RANGE 0 TO 16;
    SIGNAL a : STD_LOGIC_VECTOR(31 DOWNTO 0);

```

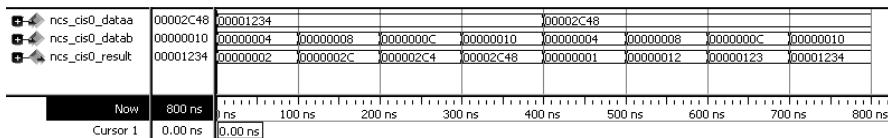


Fig. 9.39. MODELSIM simulation of the bitreverse operation

```
-- local custom instruction signals

BEGIN
-- custom instruction logic
-- note: external interfaces can be used as well
b <= CONV_INTEGER(ncs_cis0_datab);
PROCESS(ncs_cis0_dataaa, b)
BEGIN
    -- WAIT UNTIL clk ='1';
    a <= ncs_cis0_dataaa;
    ncs_cis0_result <= (OTHERS => '0');
    IF b=4 THEN -- sawp 4 bits
        FOR k IN 0 TO 3 LOOP
            ncs_cis0_result(k) <= a(3-k);
        END LOOP;
    ELSIF b=16 THEN -- swap 16 bits
        FOR k IN 0 TO 15 LOOP
            ncs_cis0_result(k) <= a(15-k);
        END LOOP;
    ELSIF b=12 THEN -- swap 12 bits
        FOR k IN 0 TO 11 LOOP
            ncs_cis0_result(k) <= a(11-k);
        END LOOP;
    ELSE -- default sawp b=8 bits
        FOR k IN 0 TO 7 LOOP
            ncs_cis0_result(k) <= a(7-k);
        END LOOP;
    END IF;
END PROCESS;

END ARCHITECTURE a_custominstruction;
```

The simulation in Fig. 9.39 shows the operation for the 4, 8, 12, and 16 bitreverse operation. The location of the half-byte change can be seen and within this half-byte we have bitreverse operation $1 \rightarrow 8, 2 \rightarrow 4, 3 \rightarrow C$, and $4 \rightarrow 2$.

To evaluate the custom instruction feature of the Nios processor, the following three implementations are compared:

- Software implementation
- Using the Altera SWAP CI
- Custom instruction for length 4-, 8-, 12-, and 16-bit data

Software implementation. A software implementation of the bitreverse operation will depend on the value a to be reversed and the number of bits b in the word. Both should be transferred to the software swap function as shown below:

```
int SW_BITSWAP(int a, int b) { int lsb, k, r=0;
    int t=a;
    for (k=0;k<b;k++)
    {
        lsb = t & 1; // take LSB
        r = r*2 + lsb; // add lsb and shift left
        t >>= 1; // shift to right by one bit
    }
    return(r);
}
```

The SW implementation then goes through the b bit in the input word and builds the reverse values starting from the LSB.

Instantiation of Altera custom logic block. Altera provides a custom 32 bitreverse block call Bitswap; see Fig. 9.40. In Qsys we can just right click it and then use the Add to add it to the CI computer system. The Altera CI is a 32 bitreverse operation and if we therefore like to use it for a b bitreverse only, then after a 32-bit bitreverse operation we need to shift the result down to the right location. In c-code that would look like

```
a_swap = ALT_CI_NIOS_CUSTOM_INSTR_BITSWAP_0(a);
a_swap >>= 32 - b; // For 32-bit type swap
```

This will require a few additional clock cycles to compute depending on the Nios processor type in use. If we want to avoid the extra shift operations then we should design a new CI that allows us to implement a bitreverse for all four designed lengths.

New CI design. After we have added our new CI to the CI computer system we will be able to do a bitreverse with a single software instruction like

```
a_swap = ALT_CI_CI_SWAP_0(a,b);
```

there is no need any more for any additional shift operation. The code for the new CI design was shown above.

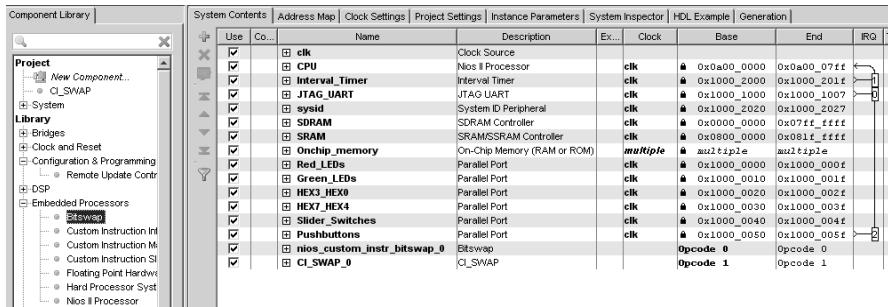


Fig. 9.40. The Altera CI library and the designed CI computer in Qsys

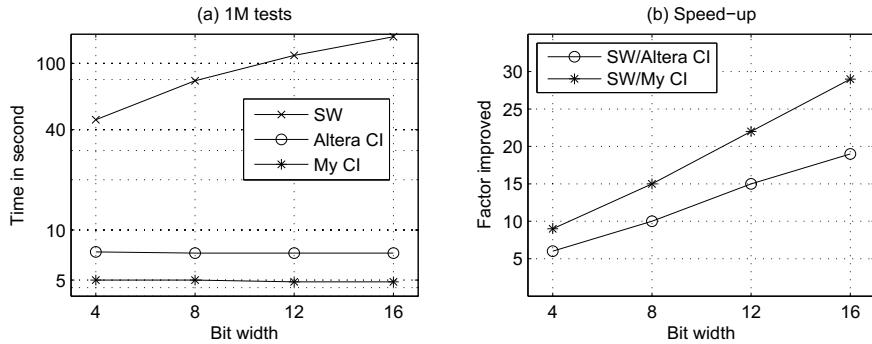


Fig. 9.41. Performance of the CI swap operations. (a) Time needed for 10^6 swap operations. (b) Speed factor improvement for the CI

Nios bitreverse performance results. The clock timer `alt_nticks()` in a Nios processor usually works with around 100 ticks per second. In order to have enough measurements the number of ticks measured should be around 100, i.e., we should use at least 1s run time for any of the algorithms. After we have measured the time it took the three algorithms for 10^6 bitreverse operations we can plot these data as shown in Fig. 9.41a.

If we now like to see the increase in speed from the CI implementation we would use a relation such as

$$\text{Speed increase} = \frac{\text{Clock cycles software only design}}{\text{Clock cycles for custom design}} \quad (9.8)$$

From Fig. 9.41b we see that the Altera CI with the additional need for the shift operation reaches a factor of 6 – 19 improvement, while our own CI without the need of the extra shift will achieve speed-up of 9 – 29 when compared with the software only realization.

The complete test bench software is on the CD as `my_swap.c` and can also be used to test the SW or CI. It has three parts: starting with a hex pattern `0x12345678` it shows the bitreverse operation for the three algorithms and

the four bit widths. In the second part the timing measurements are done and in the third part we use the LEDs and switches of the DE2 board to have a hands-on experience with the bitreverse operation:

```
#define switches (volatile short *) 0x10000040
#define     leds (short *)          0x10000000
...
b=16; while (1) { /* run forever */
    a = *switches; /* read the switch value*/
    //a_swap = SW_BITSWAP(a,b);
    a_swap = ALT_CI_NIOS_CUSTOM_INSTR_BITSWAP_0(a);
    a_swap >>=32-b; // For 32-bit Altera type swap
    //a_swap = ALT_CI_CI_SWAP_0(a,b);
    *leds = a_swap; /* Display on LEDs */
}
```

We first define the I/O addresses of LEDs and switches from the `system.h` file and specify both as 16-bit short integer. The `while` loop then runs forever: it reads the switch value, computes one of the three bitswap operations, and finally displays the results on the red LEDs.

Exercises

Note: If you have no prior experience with the Quartus II software, refer to the case study found in Sect. 1.4.3, p. 32. If not otherwise noted use the EP4CE115F29C7 from the Cyclone IV E family for the Quartus II synthesis evaluations.

9.1: Using the following categories:

- | | |
|-------------------------------------|------------------------|
| (A) Applications software | (F) Output device |
| (B) Systems software | (G) Input device |
| (C) High-level programming language | (H) Semiconductor |
| (D) Personal computer | (I) Integrated circuit |
| (E) Minicomputer/workstation | (J) Supercomputer |

classify the following examples by putting the answers in the brackets provided [] to the right:

Device driver []	GCC []	LCD display []
Wafer []	Pentium Pro []	copy (DOS) or cp (UNIX) []
MS Word []	SRAM []	Internet browser []
Microphone []	SUN []	Loudspeaker []
CD-ROM []	Metal layer []	dir (DOS) or ls (UNIX)[]

9.2: Repeat Exercise 9.1 for the following items:

Spreadsheet []	Mouse []	Silicon []
DEC Alpha []	Cray-1 []	Macintosh []
Operating system []	DRAM []	Pascal []
Microprocessor []	PowerPC []	Compiler []
Cathode ray tube display []	Printer []	Assembler []

9.3: Consider a machine with three instructions (**ADD/MUL/DIV**) with the following clock cycles per instruction (CPI) data:

Instruction class CPI for this instruction class

ADD = 1, MUL = 1, DIV = 1.

We have measured the code for the same program for two different compilers and obtained the following data:

Code from	Instruction counts for each class		
	ADD	MUL	DIV
Compiler 1	25×10^9	5×10^9	5×10^9
Compiler 2	47×10^9	18×10^9	5×10^9

Assume that the machine's clock rate is 200 MHz. Compute execution time and MIPS rate = instruction count / (execution time $\times 10^6$) for the two compilers using the following steps. Compute:

- (a) CPU clock cycles for compiler 1 =
- (b) Execution time using compiler 1 =
- (c) MIPS rate using compiler 1 =
- (d) CPU clock cycles for compiler 2 =
- (e) Execution time using compiler 2 =
- (f) MIPS rate using compiler 2 =
- (g) Which compiler is better in terms of MIPS?
- (h) Which compiler is better in terms of execution time?

9.4: Repeat Exercise 9.3 for the following CPI data: **ADD=1 MUL=5 DIV=8** and a machine clock rate of 250 MHz. We have measured the code for the same program for two different compilers and obtained the following data:

Code from	Instruction counts for each class		
	ADD	MUL	DIV
Compiler 1	35×10^9	10×10^9	5×10^9
Compiler 2	60×10^9	5×10^9	5×10^9

Answer the questions (a)-(h).

9.5: Compare zero- to three-address machines by writing programs to compute

$$g = (a * b - c) / (d * e - f) \quad (9.9)$$

for the following instruction sets:

- (a) Stack: PUSH Op1, POP Op1, ADD, SUB, MUL, DIV.
- (b) Accumulator LA Op1, STA Op1, ADD Op1, SUB Op1, MUL Op1, DIV Op1. All arithmetic operation use as the second operand the accumulator and the result is stored

in the accumulator, see (9.4), p. 647

(c) Two-operand LT Op1, M; ST Op1, M; ADD Op1, Op2; SUB Op1, Op2; MUL Op1, Op2; DIV Op1, Op2. All arithmetic operations use two operands according to (9.5), p. 649.

(d) Three-operand LT Op1, M; ST Op1, M; ADD Op1, Op2, Op3; SUB Op1, Op2, Op3; MUL Op1, Op2, Op3; DIV Op1, Op2, Op3. All arithmetic operations use three operands according to (9.6), p. 649.

9.6: Repeat Exercise 9.5 for the following arithmetic expression:

$$f = (a - b)/(c + d \times e). \quad (9.10)$$

You may rearrange the expression if necessary.

9.7: Repeat Exercise 9.5 for the following arithmetic expression:

$$h = (a - b)/((c + d) * (f - g)). \quad (9.11)$$

You may use additional temporary variables if necessary.

9.8: Convert the following infix arithmetic expressions to postfix expressions.

- (a) $a + b - c - d + e$
- (b) $(a + b) * c$
- (c) $a + b * c$
- (d) $(a + b)^{(c - d)}$ (^ power-of symbol)
- (e) $(a - b) \times (c + d) + e$
- (f) $a \times b + c \times d - e \times f$
- (g) $(a - b) \times (((c - d) \times e) \times f) / g) \times h$

9.9: For the arithmetic expression from Exercise 9.8 generate the assembler code for a stack machine using the program **c2asm.exe** from the book CD.

9.10: Convert the following postfix notation to infix:

- (a) $abc + /$
- (b) $ab + cd - /$
- (c) $ab - c + d \times$
- (d) $ab/cd \times -$
- (e) $abcde + \times \times /$
- (f) $ab + c ^$ (with ^ power-of symbol)
- (g) $abcde/f \times -g + h/\times +$

9.11: Which of the following pairs of postfix expression are equivalent?

- (a) $ab + c +$ and $abc + +$
- (b) $ab - c -$ and $abc - -$
- (c) $ab * c *$ and $abc * *$
- (d) ab^c^c and $abc^c^$ (with ^ power-of symbol)
- (e) $ab \times c +$ and $cab \times +$
- (f) $ab \times c +$ and $abc + \times$
- (g) $abc + \times$ and $ab \times bc \times +$

9.12: The URISC machine suggested by Parhami [357] has a single instruction only, which performs the following: subtract operand 1 from operand 2 and replace operand 2 with the result, then jump to the target address if the result was negative. The instructions are of the form **urisc op1,op2,offset**, where offset specifies the next instruction offset, which is 1 most of the time. Assume that all registers are initialized with -1 at reset/startup. Develop URISC code for the following functions:

- (a) clear (src).
- (b) dest = (src1)+(src2).
- (c) exchange (src1) and (src2).
- (d) goto label.
- (e) if (src1>=src2), goto label.
- (f) if (src1=src2), goto label.

9.13: Extend the lexical VHDL analysis `vhdlex.1` (see p. 665) to include pattern matching for:

- (a) Bits '0' and '1'
- (b) Bit vectors "0/1..0/1"
- (c) Data type definition like `BIT` and `STD_LOGIC`
- (d) All keywords found in HDL code `example`
- (e) All keywords found in HDL code `sqr`

9.14: Write a lexical VHDL analysis `float.1` that recognizes floating-point numbers of the type

- (a) 1.5
- (b) 1.5e10
- (c) 1.5e10 and integers, e.g., 5

9.15: Write a lexical VHDL analysis `vhdlwc.1` that counts character, VHDL key words, other words, and lines, similar to the UNIX commands `wc`. Test your analyzer with `d_ff.vhd` (p. 665) that has 17 keywords.

9.16: Write a simple lexical natural-language analyzer. Your scanner should classify each word into verbs and others. Example session:

```
user: do you like vhdl
lexer: do: is a verb
lexer: you: is not a verb
lexer: like: is a verb
lexer: vhdl: is not a verb
```

9.17: Extend the natural-language lexical analyzer from Exercise 9.16 so that the scanner classifies the items into verb, adverb, preposition, conjunction, adjective, pronoun, and noun. Examples: is, very, to, if, their, I, and boy, respectively.

9.18: Extend the natural-language lexical analyzer from Exercise 9.16 by adding a symbol table that allows you to add new words and its type to your dictionary. The type definition has to start in the first column. Example session:

```
user: verb is am run
user: noun dog cat
user: dog run
lexer: dog: noun
lexer: run: verb
```

9.19: Develop a Bison grammar `nlp` that finds a valid sentence of the form `subject VERB object`, where `subject` can be of type `NOUN` or `PRONOUN` and `object` is of type `NOUN`. For the lexical analysis use a similar scanner to that developed in Exercise 9.18. If you run the sentence parser, print out a statement if the sentence is valid or invalid. Example session:

```
user: verb like enjoy hate
user: noun vhdl verilog
```

```

user: pronoun I you she he
user: I like vhdl
nlp: Sentence is valid
user: you like she
nlp: Sentence is invalid

```

9.20: Add to the Bison grammar of `calc.y` for float number type the grammar rules for

- (a) Basic math `sqrt`
- (b) Trigonometric functions, e.g., `sin`, `cos`, `arctan`
- (c) The operations `ln` and `log`.

Example session:

```

user: sqrt(4*4+3*3)
calc: 5.0
user: arctan(1)
calc: 0.7853
user: log(1000)
calc: 3.0

```

9.21: Rewrite the Bison grammar of `calc.y` (keep `calc.l`) to have a reverse Polish or postfix `rpcalc.y` calculator. The calculator should support the operations `+-*/^`, and `^`. Verify your calculator with the following example session:

```

user: 1 3 +
rpcalc: 4
user: 1 3 + 5 * 2 ^
rpcalc: 400
user: 5 9 * 6 5 * 3 3 * 3 * - /
rpcalc: 15

```

9.22: Write a Bison grammar to analyze arithmetic expression and output three-address code. Use temporary variables to store intermediate results. At the end print the operation code and the symbol table. Example session:

```

-- input file one.c:
r=(a*b-c)/(d*e-f);

output from 3ac.exe

Intermediate code:
Quadruples            3AC
Op Dst Op1 Op2
(*, 4, 2, 3)          T1 <= a * b
(-, 6, 4, 5)          T2 <= T1 - c
(*, 9, 7, 8)          T3 <= d * e
(-, 11, 9, 10)         T4 <= T3 - f
(/, 12, 6, 11)         T5 <= T2 / T4
(=, 1, 12, --)        r <= T5

Symbol table:
1 : r
2 : a
3 : b
4 : T1
5 : c

```

```

6 : T2
7 : d
8 : e
9 : T3
10 : f
11 : T4
12 : T5

```

- 9.23:** Determine the equivalent assembler code and the values of the registers \$t0, \$t1, and \$t2 for the C program below. Assume that the C language variables A, B, and C have the following register assignments: A=\$t0, B=\$t1, and C=\$t2. Use – if the register value is unknown. Do not leave blank fields in the table. Use ADD, SUB, ADDI, or the SRL (shift right logical) instructions. Do not use SLL. The register \$zero always contains zero.

Step	C-code	Assembler instruction	\$t0	\$t1	\$t2
1	A = 48;				
2	B = 2 * A;				
3	C = 10;				
4	C = C / 4;				
5	A = A / 16;				

Instruction formats:

Addition (with overflow)	ADD	rd, rs, rt
Subtract (with overflow)	SUB	rd, rs, rt
Addition immediate (with overflow)	ADDI	rd, rs, imm
Shift right logical	SRL	rd, rs, shamt

- 9.24:** Repeat Exercise 9.23 for the following instructions:

Step	C-code	Assembler instruction	\$t0	\$t1	\$t2
1	B = 96;				
1	A = 2*B;				
3	C = 20;				
4	C = C / 8;				
5	B = B / 32;				

- 9.25:** Many RISC machines do not have a special instruction to reset a register. Show using add, addi, or subtract instructions how to reset the register \$t0. Register \$zero and the immediate value 0 should be used. The register \$zero always contains zero. Fill in the blanks in the following code:

- (a) ADD # Compute t0 = register with zero + register with zero
- (b) ADDI # Compute t0 = register with zero + 0
- (c) SUB # Compute t0 = t0 - t0;

- 9.26:** Many RISC machines do not have a special move instruction. Show using add, addi, or subtract instructions how to move the register \$s0 to \$t0, i.e., set \$t0 to \$s0. Register \$zero and the immediate value 0 should be used. The register \$zero always contains zero. Fill in the blanks in the following code:

- (a) ADD # Compute t0 = s0 + register with zero
- (b) ADDI # Compute t0 = s0 + 0
- (c) SUB # Compute t0 = s0 - 0

9.27: Determine the equivalent C-code and the values of the registers \$t0, \$t1, and \$t2 for the assembler program below. Assume that the C language variables a, b, and c have the following register assignments: a=\$t0, b=\$t1, and c=\$t2. Use – if the register value is unknown. The register \$zero always contains zero. Do not leave blank fields in the table.

Step	Instruction	C-code	\$t0	\$t1	\$t2
1	ADDI \$t2, \$zero, 32				
2	SRL \$t1, \$t2, 3				
3	ADDI \$t0, \$zero, 2				
4	SUB \$t2, \$zero, \$t0				
5	SLL \$t0, \$t0, 5				

9.28: Repeat Exercise 9.27 for the following instructions:

Step	Instruction	C-code	\$t0	\$t1	\$t2
1	ORI \$t0, \$zero, 4				
2	SLL \$t0, \$t0, 2				
3	SUB \$t1, \$zero, \$t0				
4	ORI \$t2, \$zero, 64				
5	SRL \$t1, \$t2, 3				

9.29: Determine the cache contents for the memory access sequence 2,1,2,5,1 for the following three caches with four blocks:

(a) Direct mapped cache

Address	Cache contents				
	Hit or miss	0	1	2	3
2					
1					
2					
5					
1					

(b) Fully associative (starting with first unused location)

Address	Cache contents				
	Hit or miss	0	1	2	3
2					
1					
2					
5					
1					

(c) Two-way set associative (replacing the least recently used)

Addr.	Cache contents							
	Hit or miss	Set 0	Flag 0	Set 0	Flag 0	Set 1	Flag 1	Set 1
2								
1								
2								
5								
1								

9.30: Repeat Exercise 9.29 for the following memory access sequence: 2,6,3,2,3.

9.31: For a cache with 8 KB of data and with a 32-bit data/address word width compute the total memory size using the following steps:

- (a) How many words are there in the cache?
- (b) How many tag bits are there in the cache?
- (c) What is the total size of the cache?
- (d) Compute the overhead in percentage of the cache.

9.32: Repeat Exercise 9.31 with 4 KB of data and a 32-bit data/address word width.

9.33: In Example 9.7 (p. 698) the MicroBlaze cache was discussed. Determine the number of BlockRAMs for the following MicroBlaze data: main memory 64 KB; cache size 4 KB; words per line=8.

- (a) The number of BlockRAMs to store the data
- (b) The number of BlockRAMs to store the tags
- (c) Using the data from (a) and (b), determine the maximum main memory size that can be addressed with this configuration.

9.34: In Example 9.7 (p. 698) the MicroBlaze cache was discussed. Determine the number of BlockRAMs for the following MicroBlaze data: main memory 16 KB; cache size 2 kB; words per line=4.

- (a) The number of BlockRAMs to store the data
- (b) The number of BlockRAMs to store the tags
- (c) Using the data from (a) and (b), determine the maximum main memory size that can be addressed with this configuration.

9.35: (a) Design the PREP benchmark 7 (which is equivalent to benchmark 8) shown in Fig. 9.42a with the Quartus II software. The design is a 16-bit binary up-counter. It has an asynchronous reset `rst`, an active-high clock enable `ce`, an active-high load signal `ld`, and 16-bit data input `d[15..0]`. The registers are positive-edge triggered via `clk`. The simulation in Fig. 9.42c shows first the count operation to 5, followed by a `ld` (load) test. At 490 ns a test for the asynchronous reset `rst` is performed. Finally between 700 and 800 ns the counter is disabled via `ce`. The following table summarizes the functions:

<code>clk</code>	<code>rst</code>	<code>ld</code>	<code>ce</code>	<code>q[15..0]</code>
X	0	X	X	0000
↓	1	1	X	<code>d[15..0]</code>
↓	1	0	0	No change
↓	1	0	1	Increment

- (b) Determine the registered performance `Fmax` using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M9Ks) for a single copy. Compile the HDL file with the synthesis optimization technique set to `Speed, Balanced`.

or **Area** as found in the **Analysis & Synthesis Settings** section under **EDA Tool Settings** in the **Assignments** menu. Which synthesis options are optimal in terms of size and **Registered Performance**?

Select one of the following devices:

- (b1) EP4CE115F29C7 from the Cyclone IV family
- (b2) EP2C35F672C6 from the Cyclone II family
- (b3) EPM7128SLC84-7 from the MAX7000S family
- (c) Design the multiple instantiation for benchmark 7, as shown in Fig. 9.42b.
- (d) Determine the registered performance **F_{max}** using the **TimeQuest** slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for the design with the maximum number of instantiations of PREP benchmark 7. Use the optimal synthesis option you found in (b) for the following devices:
 - (d1) EP4CE115F29C7 from the Cyclone IV E family
 - (d2) EP2C35F672C6 from the Cyclone II family
 - (d3) EPM7128SLC84-7 from the MAX7000S family

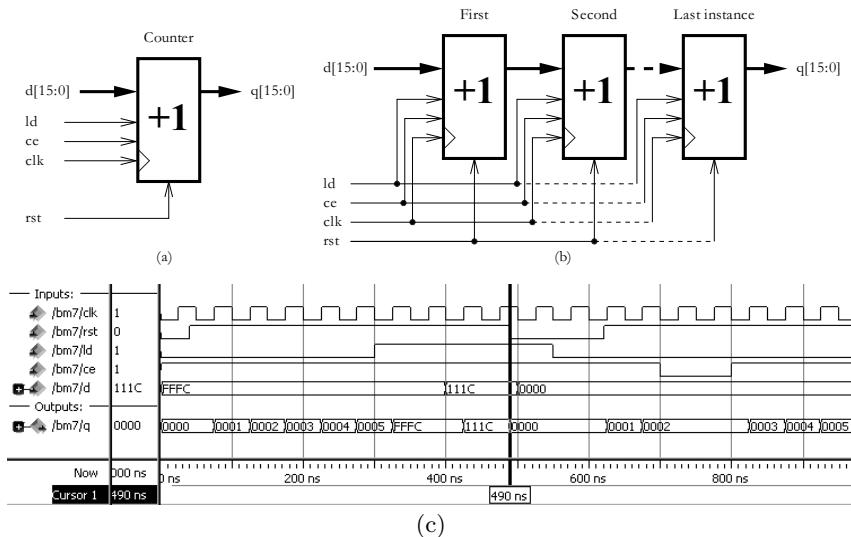


Fig. 9.42. PREP benchmark 7 and 8. (a) Single design. (b) Multiple instantiation. (c) Test bench to check the function

9.36: (a) Design the PREP benchmark 9 shown in Fig. 9.43a with the Quartus II software. The design is a memory decoder common in microprocessor systems. The addresses are decoded only when the address strobe **as** is active. Addresses that fall outside the decoder activate a bus error **be** signal. The design has a 16-bit input **a[15..0]**, an asynchronous active-low reset **rst** and all flip-flops are positive-edge triggered via **clk**. The following table summarizes the behavior:

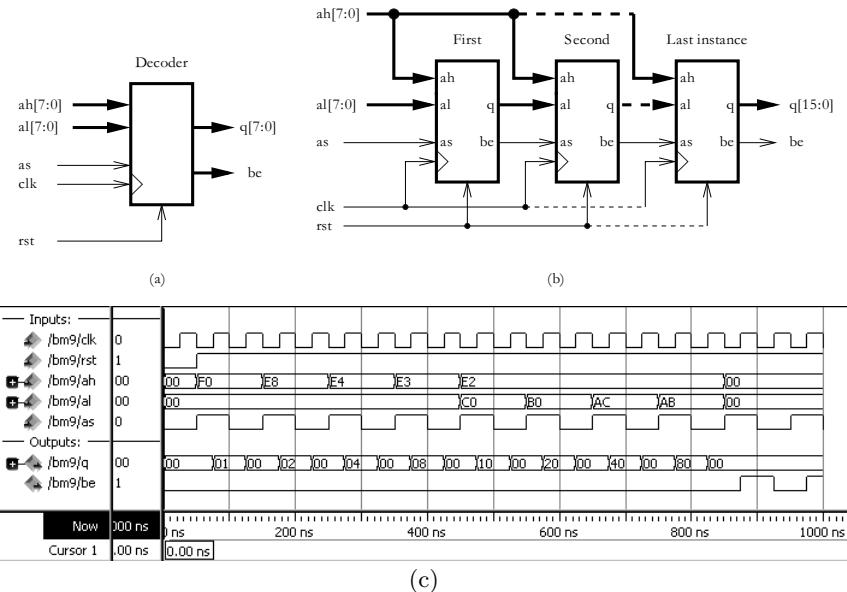


Fig. 9.43. PREP benchmark 9. (a) Single design. (b) Multiple instantiation. (c) Test bench to check the function

rst	as	clk	A (hex)	q[7..0] (binary)	be
0	X	X	X	00000000	0
1	0	✓	X	00000000	0
1	1	0	X	q[7..0]	be
1	1	✓	FFFF to F000	00000001	0
1	1	✓	EFFF to E800	00000010	0
1	1	✓	E7FF to E400	00000100	0
1	1	✓	E3FF to E300	00001000	0
1	1	✓	E2FF to E2C0	00010000	0
1	1	✓	E2BF to E2B0	00100000	0
1	1	✓	E2AF to E2AC	01000000	0
1	1	✓	E2AA	10000000	0
1	1	✓	E2AA to 0000	00000000	1

Where X is don't care. Try to match the simulation in Fig. 9.43c for the function test. Note that the original be definition requires be stored be for as=f, but the simulation shows be differently. The coding matches the simulation rather than the original truth table.

(b) Determine the registered performance **Fmax** using the **TimeQuest slow 85C** model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for a single copy. Compile the HDL file with the synthesis optimization technique set to **Speed**, **Balanced** or **Area** as found in the **Analysis & Synthesis Settings** section under **EDA Tool Settings** in the **Assignments** menu. Which synthesis options are optimal in terms of size and **Registered Performance**?

Select one of the following devices:

- (b1) EP4CE115F29C7 from the Cyclone IV E family
- (b2) EP2C35F672C6 from the Cyclone II family
- (b3) EPM7128SLC84-7 from the MAX7000S family
- (c) Design the multiple instantiation for benchmark 9, as shown in Fig. 9.43b.
- (d) Determine the registered performance F_{max} using the TimeQuest slow 85C model and the used resources (LEs, multipliers, and M4Ks/M9Ks) for the design with the maximum number of instantiations of PREP benchmark 9. Use the optimal synthesis option you found in (b) for the following devices:
 - (d1) EP4CE115F29C7 from the Cyclone IV E family
 - (d2) EP2C35F672C6 from the Cyclone II family
 - (d3) EPM7128SLC84-7 from the MAX7000S family

10. Image and Video Processing

Image and video processing can be considered as a generalization of the one dimensional (1D) signal processing to 2D and 3D, respectively. In image processing we add a second coordinate in the y direction and for video processing we would add a temporal a.k.a. spatial dimension. Many of the principals we have discussed so far like filtering or FFT can be directly extended to 2D and 3D processing and we will keep the discussion of these algorithms short. There are however several aspects in image and video processing that usually do not occur in 1D DSP. These are for instance the different kinds of median nonlinear filters used to remove impulse noise; edge detection to find discontinuities in the vision system that may be used to reduce the input features; or motion vectors in video processing that determine how a block has been moved from one frame to the next and can be used to compress the video data. Last but not least, the many algorithms used in morphological image processing do not have an equivalent in 1D DSP at all.

On the technology side we also have to realize that we now have FPGAs available with substantial on-chip memory blocks (e.g., the Stratix II with 512 Mbits blocks) such that efficient FPGA implementation of image and video processing algorithms has become feasible. Without fast on-chip memory a substantial development time had to be spent in the past to implement fast interfacing to external memory chips such as DDRAM, QDR, RLDRAM, SDRAM, etc.

There are several approaches to implement image processing algorithms and we will discuss three different approaches in later case studies: (1) direct HDL design; (2) IP library design; (3) embedded microprocessor and custom co-processor (μ P). You are also encouraged to study an image and video processing book; there are many good textbooks available today as this is a standard topic in most graduate computer engineering curricula [391–396]. An excellent hands-on start is provided in MATLAB via the image processing and computer vision toolboxes. On UNIX systems the graphic program XV has integrated many algorithms that we will discuss in this chapter [397].

However, before we go into the details of 2D and 3D processing let us first have a look at some basic differences to 1D processing, image and video formats, and a couple of introductory examples.

10.1 Overview on Image and Video Processing

One of the major differences between 1D and image processing is the fact that in 1D processing we usually insist that our filter are causal and hence realizable. In image processing we usually work with stored data and also do not want to introduce a “shift” of our data every time we apply a filter operation. As a consequence most often:

- We use “a-causal” filters where we replace the center pixel with the filter output.
- We use odd filters of equal length in x and y only, i.e., $3 \times 3, 5 \times 5, 7 \times 7$, etc.
- The filters are specified as a filter kernel and not as a convolution operation, i.e., for a filter size $L = 2N+1$; $J(r, c) = f \star I = \sum_{k=-N}^{k=N} \sum_{l=-N}^{l=N} f(k, l)I(r+k, c+l)$ is computed. In the convolution operation one signals need to be flipped before applying the matrix multiplication.

Another issue that is of greater concern in 2D than in 1D is the border effect. For a filter of mask of size $L \times L$ and an image of size $R \times C$ we would lose $2RL/2 + 2CL/2 - 4(L/2)^2 = L(R+C) - L^2$ pixels from the border. For instance, if a Laplacian-of-Gaussian (LOG) filter with $\sigma = 3$ gives a kernel size of $L = 33$ and a 256×256 image it will lose 24% of the image area due to border effect [398]. IIR filters perform better in this regard, but are less frequently used due to the nonlinear phase and possible stability problems of IIR filters. Typically we try to minimize the bordering effect by extending the image by half the filter size via:

- A constant value, typically $x = 0$
- A symmetric extension, e.g., ...3, 2, 1|1, 2, 3, ...
- Replicating the border value, e.g., ...1, 1, 1|1, 2, 3, ...
- Circular (a.k.a. wrap around) extension of the image, e.g., ..., 8, 9, 10|1, 2, 3..., 8, 9, 10|

These extensions have to be done in row and column directions.

From a hardware perspective we typically use something similar to the circular approach, just with one line offset, since the image data are most often arranged in a one-dimensional array, i.e.,

...	1	2	3	4	5	10	11	12	...
3	4	5	10	11	12	13	14	...	

The end of the first line (values 3,4,5) is extended by the values starting at the begin of the second line, and for the beginning of the second line (values 10,11,12) the end value of the first line are used in the computation. This approach works well only if we assume that the border of the image has a background and that the background typically is similar across the whole image.

10.1.1 Image Format

A single point in a digital image is usually called a pixel or pel, where typically 8 bits of data are used, (up to 12 bits in medical imaging). Black is usually coded with zero, and white with maximum value, i.e., 255 in an 8-bit format. A full color spectrum can be generated with just three basic colors: red, green, and blue (RGB). For a transformation from gray to RGB all three are assigned the same gray value. The eye is usually more sensitive to gray changes than color changes and coding schemes therefore use a gray value (luminance) and differential color coding (chrominance). The luminance is transmitted with more precision (bits) and/or resolution (pixels) than the chrominance without degrading the color image quality. In RGB all three have the same number of bits. To transform between RGB and the differential schemes there are two methods (YIQ and YUV a.k.a. YCrCb) popular in TV, JPEG and MPEG. In NTSC TV systems the YIQ coding uses the following transform:

$$\begin{aligned} \text{Luminance: } & Y = 0.2989R + 0.5870G + 0.1140B \\ \text{R-cyan: } & I = 0.5959R - 0.2744G - 0.3216B \\ \text{magenta-green: } & Q = 0.2115R - 0.5229G + 0.3114B \end{aligned} \quad (10.1)$$

For the YCrCb scheme, as suggested by ITU recommendation BT.601-5, the following equations are used:

$$\begin{aligned} & Y = 0.299R + 0.587G + 0.114B \\ C_r = & 0.713(R - Y) = 0.500R - 0.419G - 0.081B \\ C_b = & 0.564(B - Y) = 0.169R - 0.331G + 0.500B \end{aligned} \quad (10.2)$$

and for 8-bit quantization and the constraint that the luminance range is 16 to 220 (to provide a working margin) and maximum chrominance is 225 with zero level at 128 the ITU suggest the following mapping:

$$\begin{aligned} Y &= \frac{77}{256}R + \frac{150}{256}G + \frac{29}{256}B \\ C_r &= \frac{131}{256}R - \frac{110}{256}G - \frac{21}{256}B + 128 \\ C_b &= \frac{44}{256}R - \frac{87}{256}G + \frac{131}{256}B + 128 \end{aligned} \quad (10.3)$$

with the Altera video processing toolbox a similar but slightly modified (luminance range 16 to $(0.257 + 0.504 + 0.098) \times 255 + 16 \approx 235$, chrominance range $0.439 \times 256 + 128 \approx 240$) approximation is used:

$$\begin{aligned} Y &= 0.257R + 0.504G + 0.098B + 16 \\ C_r &= 0.439R - 0.368G - 0.071B + 128 \\ C_b &= -0.148R - 0.291G + 0.439B + 128 \end{aligned} \quad (10.4)$$

Traditionally images have been enumerated as a cathode ray tube (CRT) type monitor would display the pixels, i.e., the pixels would start from the

upper left corner and then proceed for a row from left to right and then start with the next line again from the left. An image is then specified by the *resolution* like 640×480 , 800×600 , 1024×768 , or 1280×1024 , given by the number of pixels in the *x* (horizontal) first followed by the number of rows in the *y* direction, i.e.,

(0,0)	(0,1)	...	(0,C-1)
(1,0)	(1,1)	...	(1,C-1)
:		..	:
(R-1,0)	(R-1,1)	...	(R-1,C-1)

In video processing the typical image size is a multiple of 16×16 macro blocks and the most popular sizes are Common Intermediate Format (CIF) of size 352×288 and Quarter CIF (QCIF) of size 176×144 .

In color image processing each pixel would represented a red, green, and blue (RGB) pixel. While this enumeration is used often not all image formats follow it. The popular raw BMP format for instance starts in the lower left corner and finishes at the last pixel in the upper right. There are several image formats in use today. Most formats use 8 bits for gray scale data and also 8 bits for each of the RGB data. A typical image format starts with a header that includes all necessary information such as rows, columns, color table, compression scheme, etc., and then the image data follow. Some of these formats allow substantial data compression (e.g., GIF and JPEG) that can be lossless or lossy. Only a few still image formats have been defined by international committees such as ISO/IEC/ITU for JPEG and JPEG-2000; many other were defined by companies and the IP rights are not always 100% clear. Let us start with some popular non-standard formats and then move onto the standard formats. All formats discussed are supported by MATLAB in the `imread()` function.

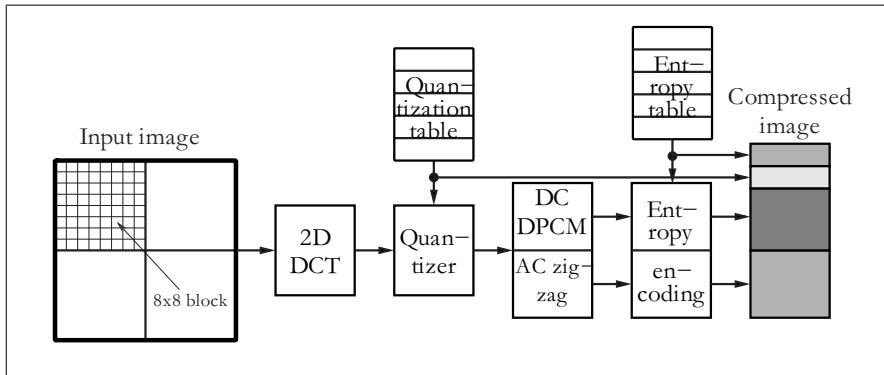
The BMP format is a raw image format that does not include data compression. It uses a color table for RGB with 8 bits for each component. It also allows one to use less than 24 bits for color coding. You can code an 8-bit color table that only includes the most often used colors and store this table in the file header. For 8-bit gray scale no color table is needed. This format is used in the DE2 demonstrations like the `DE2_Default` project and is also popular in embedded μ P designs [391]. The logic book [63, App. F] includes a tutorial on the `BMP2MIF` converter that comes with the DE2 boards.

The graphic interchange format (GIF) format includes lossless substantial image compression and is usually a very good choice when designing PPT slides. It has also been used to make small movies a.k.a. GIF animation.

Portable Network Graphic (PNG) format is another lossless compression scheme that is often used for WWW pages. The PNG format uses differential encoding but does *not* use the Lempel–Ziv–Welch (LZW) compression algorithm like GIF and is therefore free from any licences fees for the LZW method.

Table 10.1. Overview on image formats

Format	Int. STD	Lossless. compression	Lossy compression	MATLAB
BMP	—	—	—	✓
GIF	—	✓	LZW	✓
PNG	—	✓	DPCM	✓
JPEG	✓	✓	DCT	✓
JPEG-2K	✓	✓	DWT	✓

**Fig. 10.1.** The JPEG compression scheme

Since the JPEG and JPEG-2000 formats are the only official standards for still images, let us have a closer look at these formats. The JPEG standard, also known as ITU-T recommendation T.81, is available for free download from the CCITT webpage [399]. Figure 10.1 shows the most important coding and decoding step of the baseline DCT-lossy compression.

Note, however, that only the decoder is specified in the standard, leaving the encoder open for possible improvements. The input image (assumed to be gray scale; for color image the RGB components are typically first transform using YIQ (10.1) or YCrCb (10.2)) having 8-bit data is first divided into 8×8 non-overlapping blocks and (forward) DCT transformed. Since the type two 2D DCT is used, the 2D transforms are separable in x and y directions and we apply first eight row and then eight point column transforms [206]. The DC value is found in the upper left corner of our 8×8 data block. Then the luminance and chrominance DCT coefficient are quantized with different tables. As an example for the luminance, a suggested quantization table, is given in the standard [399, p. 143]; see Fig. 10.2a.

The values shown are the coefficients needed to reconstruct the DCT values at the decoder side by multiplication. The quantization on the encoder is done by first dividing by the value shown in Fig. 10.2a followed by rounding

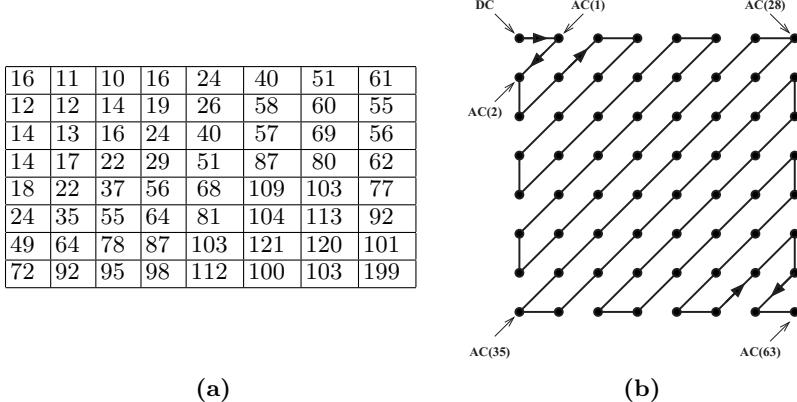


Fig. 10.2. The JPEG compression scheme details. (a) Quantization table. (b) Zig-zag encoding of AC coefficient

to the nearest integer, moving the computation burden (division is usually expensive) to the encoder. From the given values we see that lower frequencies have lower quantization factors, and are therefore represented with more bits. The high frequency values should come out typically as 0, or ± 1 for 8-bit input data. Using a zig-zag ordering the encoder arranges the first couple of non-zero coefficients and terminates the block with an end-of-block (EOB) symbol; see Fig. 10.2b. Since DC values (i.e., the average of all pixels in the 8×8 block) of neighbor blocks are usually similar, these are differentially encoded ($\text{DIFF} = \text{DC}_i - \text{DC}_{i-1}$) and then Huffman encoded. First we code the size with 4 bits and then the difference value, i.e.,

4-bit SSSS	DIFF values
0	0
1	-1,1
2	-3,-2,2,3
3	-7,...,-4,4,...,7

All other 63 AC coefficients are encoded with a variable run-length coding (VLC) of the format (run length of zeros, next value), each using 4 bits. If a run is longer than 15 then an additional special symbol (15,0) is introduced; (0,0) is coding for the EOB symbol. For instance, the DCT coding coefficients

$$(10, 3, 0, 0, -1, 1, 0, 0, 0, 2, \text{EOB}) \quad (10.5)$$

would first code the DC based on the previous DC block value and the AC coefficients would be coded as

$$(0, 3) \quad (2, -1) \quad (0, 1) \quad (4, 2) \quad (0, 0). \quad (10.6)$$

The DCT values can also be further compressed using a Huffman coding. Appendix K of the standard provides example luminance and chrominance

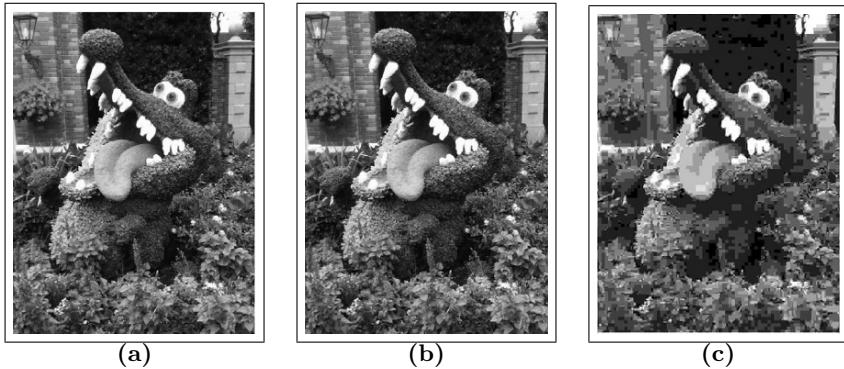


Fig. 10.3. JPEG compression results. (a) Original BMP (677 KB) image. (b) Q20 low compression JPEG (53 KB) image. (c) Q01 high compression JPEG (22 KB) image

DC and AC tables that need to be included in the JPEG file header. The inclusion of the coding tables is usually the reason why the reduction in file size from medium to high compression is less than expected. Figure 10.3 shows some example JPEG coding. For low compression ratios the difference between JPEG and BMP is hardly visible. For higher compression the JPEG shows blocking effects due to the DCT blocks.

The JPEG-2000 standard may be considered as an update of the original JPEG standard, but has some substantial changes compared to JPEG. Most important the DCT 8×8 block processing is replaced by a DWT processing typically using Daubechies length 9/7 floating-point filters for lossy and 5/3 integer for lossless compression. Quantization and arithmetic coding follows the initial DWT; see Fig. 10.4. JPEG-2000 was a joint effort of several groups (ISO, IEC/ITU) but unfortunately the standard specification documentation [400] is not available free and is still for sale at (96 CHF) at the time of writing. However, there are many publications available that provide enough detail [401–403] to understand the underlying ideas. The new standard is not a single algorithm, but rather a collection of algorithms like a toolbox that may or may not be used in the compression of a still image. Among the most interesting features are:

- Superior performance at low bit-rate since the DWT avoids the blocking effect of the “classic” JPEG.
- Lossless and lossy compression.
- Region-of interest coding that allows a better coding of part of the image that is considered specially important, e.g., a speaker face compared to the background.
- Robustness to bit-errors through synchronization markers, fixed length entropy coder, and repeated headers.

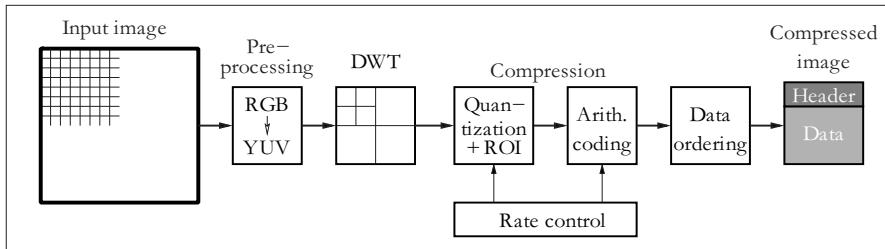


Fig. 10.4. The JPEG-2000 compression scheme

- Protective image security by means of watermarking, labeling, stamping, and encryption.

JPEG-2000 usually produces better peak signal-to-noise ratio (PSNR), mean square error (MSE), or structural similarity (SSIM) at the cost of higher complexity than JPEG. JPEG-2000 is used if the application needs either high quality or low bitrate, or one of the other features provided by JPEG-2000 such as security.

10.1.2 Basic Image Processing Operation

Among the basic image processing operations in an image and video processing toolbox we may need operations like:

- **Scale** of an image to increase or decrease the pixels processed.
- **Crop** of a region-of-interest (ROI) of the image.
- **Rotate** an image.
- **Arithmetic** to add, subtract, multiply, divide, and average of one or more images.
- Color to gray and gray to color conversions.
- Reading and storing images in data formats such as BMP, GIF, or JPEG.

Transforming single pixels based on their value is called a *point operation*. This can be an image adjustment from too much or too little overall lighting, or maybe based on the local or global pixel statistics, i.e., the image histogram. A point operation used by standard CRT monitors is called γ correction. For 8-bit data it uses the following transform:

$$y = 255(i/255)^{1/\gamma}. \quad (10.7)$$

For $0 < \gamma < 1$ we would get an exponential curve and dim the image, while for $\gamma > 1$ we get a logarithmic curve which will brighten the image; see Fig. 10.5. $\gamma = 2.2$ is used in NTSC TV systems [63, 395].

The *filtering* of an image can be either linear or nonlinear but involves not just a single pixel but also a couple of neighboring pixels. Most of these operations are used in image enhancement. Most linear methods can also

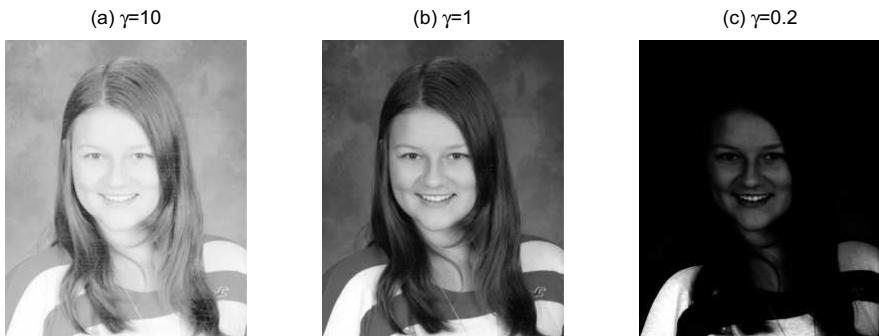


Fig. 10.5. γ Correction examples

be solved in the transform domain, such as FFT or DCT. Here are a few examples:

- Removing random noise with a lowpass filter.
- Detecting edge with a differential filter.
- Maximum filter to remove pepper (since black has code 0).
- Minimum filter to remove salt noise (since white is coded as 255).
- Median filter to reduce the salt *and* pepper noise.
- Deblurring filter to remove blurring that may occurs from a moving camera done with deconvolution filtering.
- Adaptive Wiener filtering to remove noise.

Image compression is also a field with many research activities both still images as well as video compression. Many methods take advantage of physiological effects, like the higher sensitivity to low frequency change than high frequencies. These transform codings are usually combined with a (lossless) entropy coding such as variable length coding, Huffman coding, or arithmetic coding. Most popular 2D transforms used in image and video processing are FFT, DCT, and DWT, but others like Haar transform or Hartley transform have been used.

Morphological operations are usually used to enhance the information or reduce the input space for a pattern recognition system. It is based on operations like “adding pixels in a neighborhood” (thickening, a.k.a. dilation) or “removing pixels in a neighborhood” (thinning, a.k.a. erosion). The dilation will produce more pixels and therefore make a line thicker or close small holes. The erosion will make a line thinner and remove small “noisy” regions like single pixels from the image. Often dilation and erosion are combined. Erosion followed by dilation is called *opening* and dilation followed by erosion is called *closing*. Closing and opening can be used to smooth an object, where the closing also fills small holes and gaps.

The operations can be defined for gray as well as black-and-white (BW) images, where BW is more common. For gray scale images the morphological

operations are typically maximum or minimum operations within a neighborhood; for BW images typically a neighbor mask is defined as to how pixels have to be substituted.

Figure 10.6 shows a few morphological operations using the MATLAB functions `bwmorph` and `bwperim`. The opening operation reduces the background line of the handwritten text; see Fig. 10.6b. The skeleton is a sequence of erosions and openings and produces a centerline for each object such that the structure of all objects becomes visible, just like the bones in an X-ray picture; see Fig. 10.6c. The boundary extraction can be computed via first erosion of the image with a mask M followed by an image subtraction, i.e., $BW = I - (I \ominus M)$, where \ominus represents the erosion operation. The result of this operation produces a boundary or perimeter picture and is shown in Fig. 10.6d.

Other morphological operations that have been found helpful in edge detection are the “majority black” operation that sets a pixel to black if the majority of pixels in a neighborhood are black, similar to a dilation operation.

10.2 Case Study 1: Edge Detection in HDL

Most of the information about the objects within an image is gained from the edges within the image. Finding the edge is not too difficult since different objects or the object and the background have different intensities and we just need to find the points of discontinuity in the pixel values and mark these as our edge. To detect the contrast changes we have basically two choices. We can compute the derivatives, i.e.,

$$f'(x) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}} \quad (10.8)$$

and determine the (absolute) maxima. Alternatively, we can compute the second derivative, $f''(x) = df'(x)/dx$ and then look for the zero crossing in the signal. Let us have a look at a simple one-dimensional signal and then later generalize it to 2D. Since we have pixels, we would use a mask or kernel to do the processing. For the first derivative we may use a mask like $M_1 = [-1, 1]$. For a second order derivative [404] we may use a mask like $M_2 = [1, -2, 1]$. Let us analyze a short test sequence with the first mask to convince us that such a short mask indeed works for the task at hand:

I	10	10	10	40	40	30	20	10	0	0	30	0
$I \star M_1$	-	0	0	30	0	-10	-10	-10	-10	0	30	-30
$ I \star M_1 $	-	0	0	30	0	10	10	10	10	0	30	30

The test sequence has an upward edge followed by a downward ramp, and a single “noisy” pixel at the end. We see that the edges and ramps are detected but also the sensitivity to noise of the short mask.

For the correlation with the second derivative masks $[1, -2, 1]$ we get

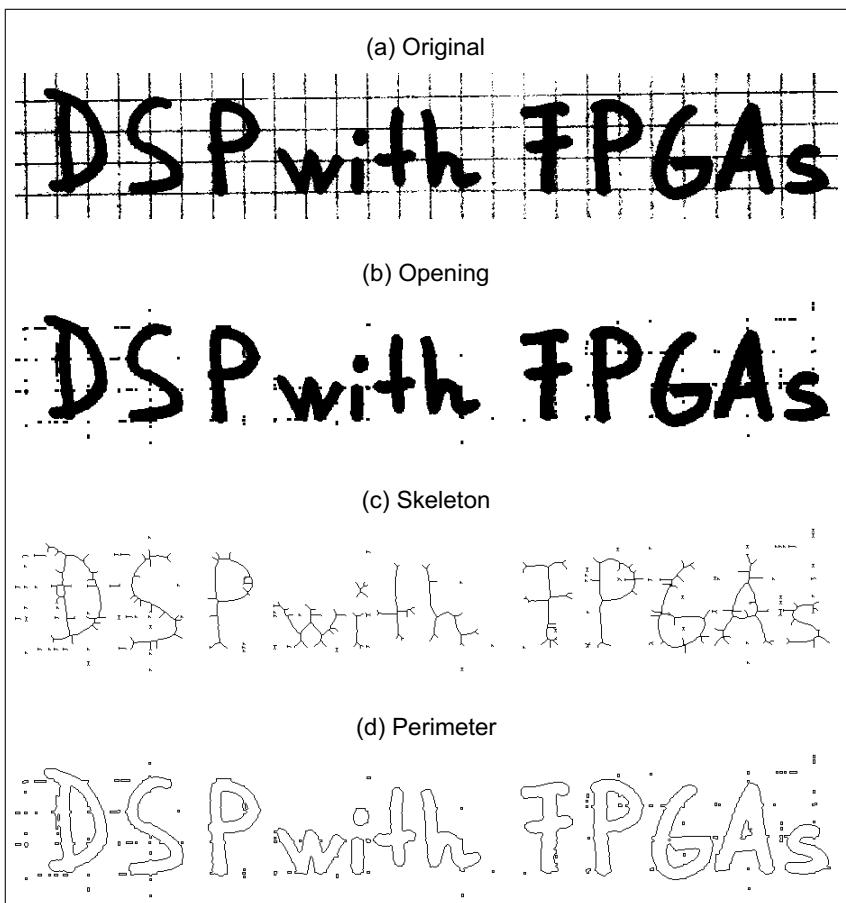


Fig. 10.6. Example of morphological operations. (a) The original image. (b) Opening operation. (c) Skeleton operation. (d) Perimeter operation

I	10	10	10	40	40	40	30	20	10	0	0	0
$I \star M_2$	—	0	30	-30	0	-10	0	0	0	10	0	—
zero-crossing	—	—	✓	✓	—	✓	—	—	—	✓	—	—

We see a \pm transition in the second derivative for the edge and the beginning of the ramp and less precision at the end of the ramp. Detecting zero-crossings somehow seemed to be more work than the detection of the maxima in the first derivative and we will implement the latter method.

For a more centered approach to avoid a shift in location in image processing, a mask such as $M = [-1, 0, 1]/2$, would be used. The factor $1/2$ comes from the worst case growth of $\sum_k |f_k| = 2$ and the desire to avoid an overflow in the 8-bit arithmetic. To reduce the noisy influence of single pixels, typically an averaging over several lines is used. This will then give

3×3 kernels such as the Prewitt masks M_x and M_y that can detect vertical and horizontal lines, respectively [405].

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad M_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \quad (10.9)$$

Another mask suggested by Sobel [406] emphasize more the center coefficient and looks as follows:

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad M_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad (10.10)$$

So far we have not decided on an appropriate scaling factor for the Sobel mask. We may choose one (pessimistic) that avoids any bit growth via $1/\sum_k |f_k| = 1/8$. Alternatively we can choose to approximate the true derivative for a horizontal or vertical edge $\dots, 0, 0, 1, 1, \dots$ and then the scale factor should be $1/4$. For the edge detection we skip the scaling as long as the data are in the valid range since we are only interested in the maxima. The correlation of our image I with our mask produces the derivatives $G_x = I \star M_x$ and $G_y = I \star M_y$. The general derivative would be the vector sum $\vec{G} = \vec{G}_x + \vec{G}_y$. Since we are only interested in the length of this vector we would compute

$$|G| = \sqrt{|G_x|^2 + |G_y|^2}. \quad (10.11)$$

The square root computation is usually too cumbersome and we can use any of the magnitude approximations discussed in Sec. 2.10 (p. 165). The standard approximation used in many papers via $|G| = |G_x| + |G_y|$ will have a large error margin of 41% since $\sin(x) + \cos(x) = \sqrt{2} \sin(x/2)$. We will use the L_1 approximation with $|G| = \max(|G_x|, |G_y|) + \min(|G_x|, |G_y|)/4$ which reduces the error margin by a factor of 3.

Let us now have a first look how this edge detectors work in practice. Figure 10.7 show an Eifel tower model and we apply the Sobel operator. First the $|G_y|$ shows the horizontal gradient, while $|G_x|$ shows the vertical gradient. The sum then shows both. A final threshold is applied to classify for edge and non-edge. The threshold in general will depend on the S/N ratio of the image. For an image with high S/N the recommendation is to use about 10-20% of the maximum gradient measured [392, 407].

For the second derivative the Laplacian of the Gaussian (LOG) filter is usually used. The Gaussian function is easy to differentiate, reproduces, and can detect edges and perform smoothing at the same time. The equations needed are

$$g(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \quad (10.12)$$

$$g'(x) = \frac{-x}{\sigma^2} g(x) \quad (10.13)$$

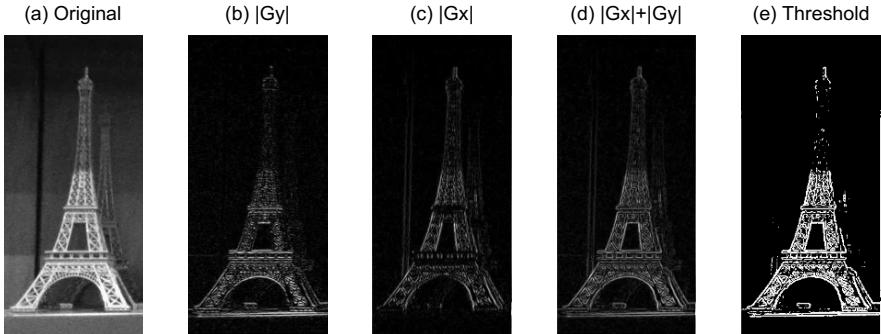


Fig. 10.7. Example of edge detection using the Sobel mask. (a) The original image. (b) Filter with M_y gives horizontal edges/details. (c) Filter with M_x gives vertical edges/details. (d) Sum of gradients. (e) Thresholding

$$g''(x) = \left(\frac{x^2}{\sigma^4} - \frac{1}{\sigma^2} \right) g(x). \quad (10.14)$$

In image processing the LOG is often approximated by the difference of Gaussian (DOG) that looks like

$$g''(x) \approx c_1 e^{-\frac{x^2}{2\sigma_1^2}} - c_2 e^{-\frac{x^2}{2\sigma_2^2}} \quad (10.15)$$

and seems to be a little easier to compute than the true LOG. Some typical masks used for the LOG are

$$M_{3x3} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad M_{5x5} = \begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix} \quad (10.16)$$

Beside this short mask more elaborate methods have also been developed, that are more robust in a low S/N environment. The Canny method [408,409] for instance first uses a smoothing filter to remove noise and then a fine classification scheme within a larger mask to select just one edge within the region. This is similar to the thinning operation used in morphological operations. The goal is to show only one line for an edge or a ramp. Figure 10.8 shows that the Canny operator detects more details than LOG or Sobel.

10.2.1 2D HDL Filter Design

Many filter architectures exist for 1D filters and introducing an additional dimension will therefore increase the number of possible architectures substantially. However, before talking about possible architectures it is useful to remember the concepts we have already discussed. When implementing FIR filters we take advantage of the ideas from Chaps. 3 and 4 on 1D filters, i.e.:

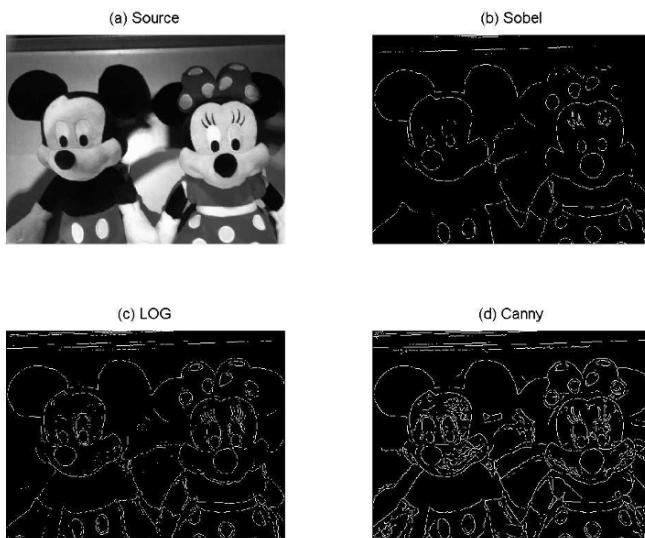


Fig. 10.8. Comparison of edge detection methods. (a) The original image. (b) Sobel operator. (c) Laplacian of Gaussian. (d) Canny detector

- Whenever possible implement the filters in the transposed form.
- Combine as many coefficients as possible in a multiplier block.
- Minimize the memory I/O, read data only once by using a tapped delay line (TDL) for the data already read.

The third concept will require that we use the $N - 1$ line or row buffers (implemented as FIFO in hardware) for an $N \times N$ mask. The row buffer should be implemented by an embedded memory block; otherwise a large number of LEs would be required to build this FIFO. We can then build an FIR filter with a single multiplier block as shown in Fig. 10.9a. However, there are a few drawbacks of the one MCM block architecture. First the output of the first row FIR is the input to the row buffer, which most likely uses more bits than the input data; hence the FIFO row buffer needs to have a wider data bus. The second drawback of the architecture come from the reuse of the row buffer for multiple filters. In the edge detection we essentially have two filters and if we slightly rearrange the filters then we can use the same row buffer for both filters, but also reduce the single MCM block to N blocks. This second architecture is shown in Fig. 10.9b again for 3×3 kernel.

Sometimes with a special set of coefficients an important simplification of the 2D filtering becomes viable. This is the case if the row and column filter are *separable*, i.e., the filters are the outer products of two 1D filters; see Fig. 10.10. For any matrix of rank one we know that the matrix can be

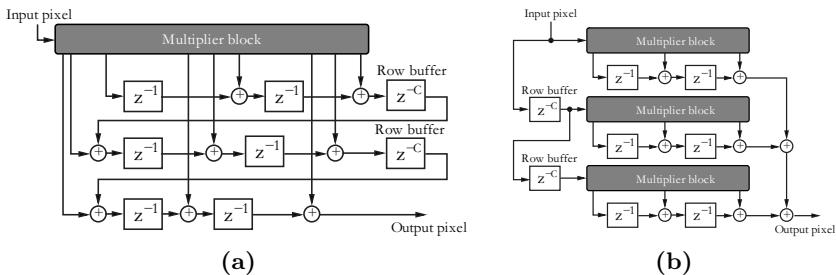


Fig. 10.9. Comparison of 2D filter methods. z^{-C} is a delay over all (remaining) columns of the entire image. (a) Single MCM filter block architecture. (b) N-MCM filter block architecture

expressed as an outer product. However, it is not always easy (or possible) to calculate the 1D filter from the 2D kernel, but if we find such a set then the filter operation can be further simplified, since we just perform a row followed by a column filtering. From our kernels discussed earlier we can see that Prewitt and Sobel are separable but the 3×3 LOG filter (10.16), p. 751, is not. Take for instance the Sobel filter from (10.10), p. 750; for M_x we compute

$$M_x = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times [-1 \ 0 \ 1] = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad (10.17)$$

and we can implement the M_x filter by first using a vertical filter mask $[1, 2, 1]^T$ followed by a horizontal $[-1, 0, 1]$ filter mask. For the LOG filter things are more complicated. We can compute $g''(x)$ as in (10.12) and then build an outer product, but this will not be the true 2D LOG filter. We can compute the LOG easily with the MATLAB function `fspecial('log', width, sigma)`; however the true 2D is usually not of rank one.

The vertical filter will no longer have a single MCM block, but the horizontal filter will have a single MCM block. Let us just briefly count the number of operations for both filter types. Remember a 1D filter of length L needs L multiplications and $L - 1$ additions per input sample. If the filter is symmetric the arithmetic count is further reduced to $(L + 1)/2$ multiplications and $L - 1$ additions assuming we have an odd filter length. Table 10.2 gives the operation count for both cases [410].

For the edge detection application another important consideration is that we can use the row buffer for both filter M_x and M_y . Figure 10.10b shows the architecture for such a two filter design.

10.2.2 Imaging System Design

After we have decided on the algorithm we try to implement the next step which would be to think about possible HW designs. Of course we will take

Table 10.2. Operation count for separable and non-separable filters for $L \times L$ filter mask for L odd

Operation	Non-separable		Separable	
	Non-symmetric	Symmetric	Non-symmetric	Symmetric
Multiplications	L^2	$L(L+1)/2$	$2L$	$2(L+1)/2$
Additions	$L^2 - 1$	$L^2 - 1$	$2(L-1)$	$2(L-1)$

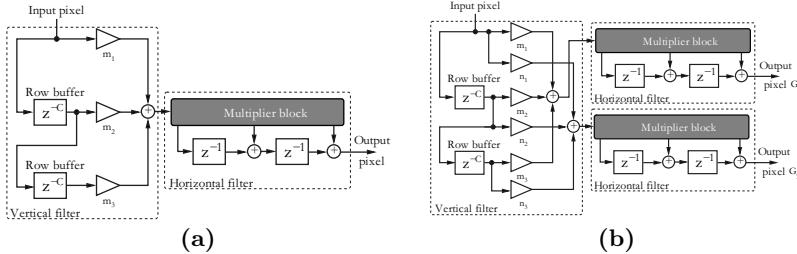


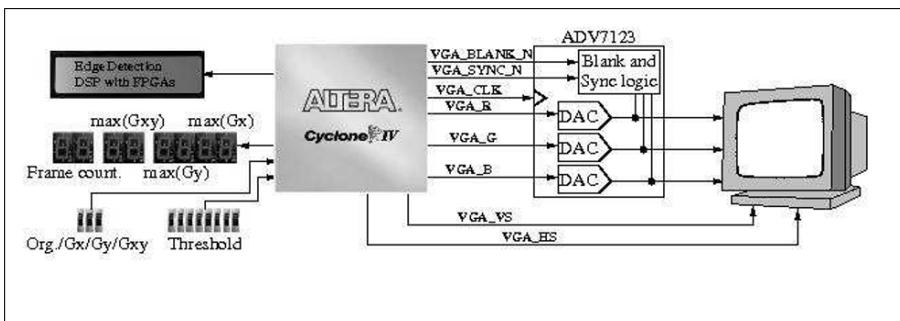
Fig. 10.10. 2D separable filter architectures. (a) Row and column filter. (b) Two filter architecture using the same row buffer

advantage of one of the development boards that usually come with one or more display options. The Xilinx Altys for instance comes with 2×2 ports for the high definition multimedia interface (HDMI) that can be used for video and audio, while the Altera DE2-115 has a TV decoder, a video graphics array (VGA), and a digital visual interface (DVI) display option. The DVI is the interface used most often for LCD monitors, while the VGA interface is one of the classic standards, still used by most PCs as a default start-up interface to display system information on a CRT monitor. The basic resolution is 640 columns and 480 rows with a refresh rate of 60 Hz. We will use this basic VGA mode since it is supported by most monitors; even LCD monitors usually have a VGA input. This will allow us to use our DE2-115 directly with a CRT monitor without buying any new display HW.

The basic system setup is shown in Fig. 10.11. The FPGA provides the necessary control and RGB data signals. Note that the DE2-115 board uses 8-bit RGB data rather than the 10-bit data of the DE2 since dealing with 10-bit data was too cumbersome and most images have 8-bit RGB resolution anyway. The VGA chip used on the DE2-115 is the Analog Devices ADV7123KSTZ140 that includes triple high speed 10-bit DACs and can run with up to 140 MHz pixel rate. We can therefore use VGA (640×480), SVGA (800×600), XGA (1024×768), or SXGA (1280×1024) display modes. Table 10.3 shows the modes and the required pixel clocks (total rows \times columns \times frame rate) and gray scale image memory requirements (color needs three times as much), i.e., rows \times columns \times 8.

Table 10.3. VGA resolution characteristic data

Mode	Refresh rate Hz	Resolution columns \times rows	Total C \times R	Pixel clock MHz	Size Mbits
VGA	60	640 \times 480	800 \times 525	25.175	2.4
VGA	85	640 \times 480	832 \times 509	36	2.4
SVGA	60	800 \times 600	1056 \times 628	40	3.84
SVGA	75	800 \times 600	1056 \times 625	49	3.84
SVGA	85	800 \times 600	1048 \times 631	56	3.84
XGA	60	1024 \times 768	1344 \times 806	65	6.29
XGA	75	1024 \times 768	1312 \times 806	75	6.29
XGA	85	1024 \times 768	1376 \times 808	95	6.29
SXGA	60	1280 \times 1024	1688 \times 1066	108	10.49

**Fig. 10.11.** The overall edge detection system design

The overall timing of a VGA signals consists of the video data and additional synchronization time since we need the CRT beam to be allowed to return to the begin of the line or the first row. We assume that external synchronization is used and not embedded synchronization using the green video channel. The line in the VGA then starts with an active low horizontal synchronization impulse. Let us have a looks at the VGA signals at 60 Hz. The synchronization is $3.8 \mu\text{s}$ long, and the pixel clock of 25 MHz turns out to be 96 clock cycles. Next comes the horizontal back porch of $1.9 \mu\text{s}$ or 48 cycles. Then for 640 clock cycles or $25.4 \mu\text{s}$ the video signal is present, followed by the front porch signal of $0.6 \mu\text{s}$ or 16 clock cycles that allows the CRT beam to return to the beginning of the line. Overall it took $96 + 48 + 640 + 16 = 800$ clock cycles for a single line. For the vertical timing we have a similar specification. Again let us look at the VGA signals at 60 Hz. The vertical synchronization impulse takes two lines, followed by the vertical back porch of 33 lines. Then the next 480 lines contain the video signal and finally ten lines of the vertical front porch allow the CRT beam to return to the first line. Figure 10.12 shows

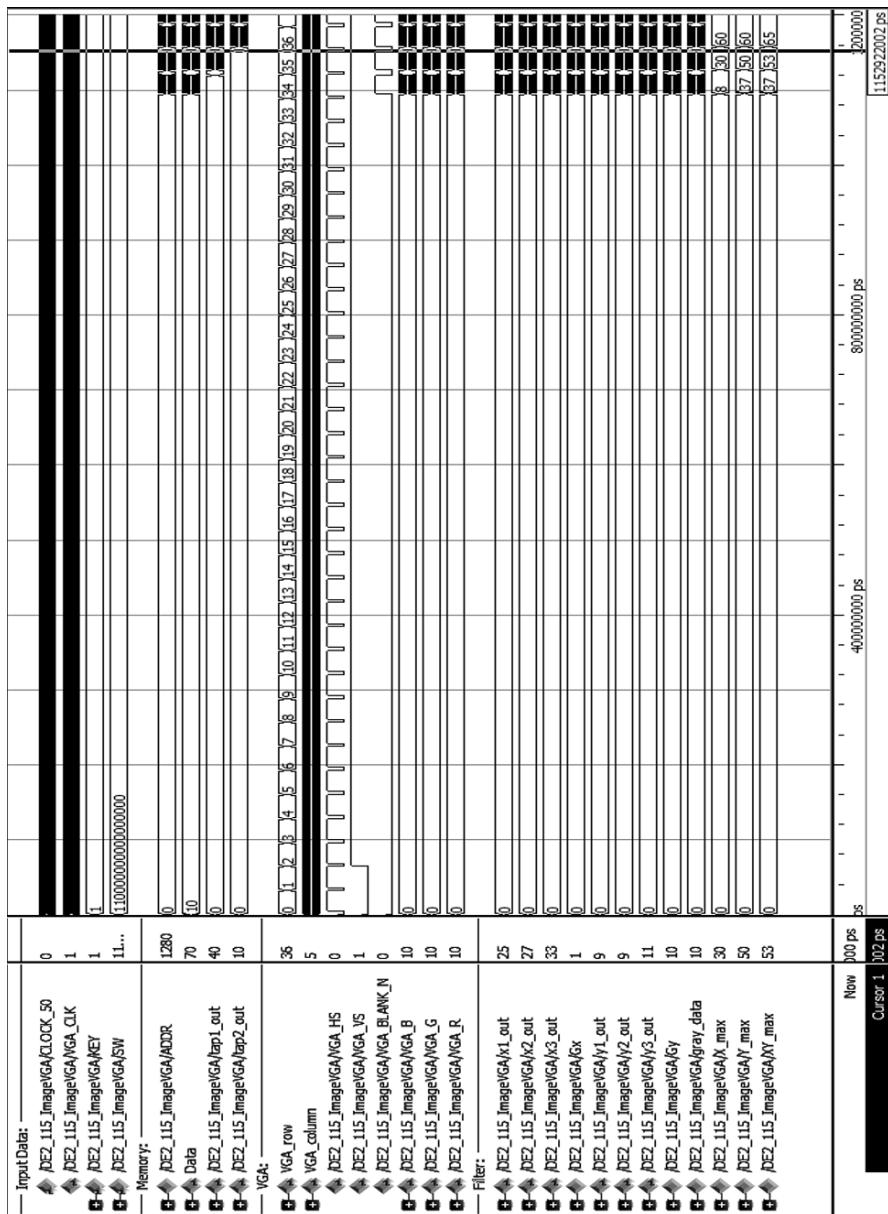


Fig. 10.12. The overall VGA MODELSIM simulation

the initial MODELSIM simulation with two lines vertical synchronization and 33 lines back porch before the video display starts.

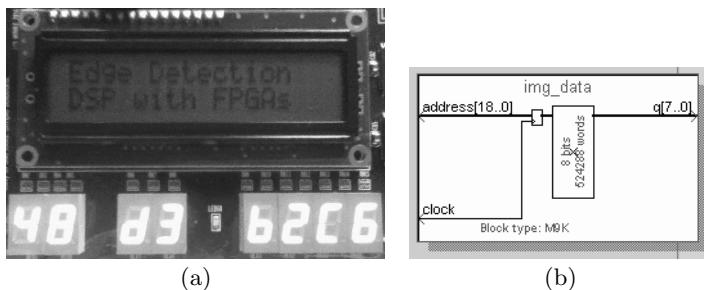


Fig. 10.13. (a) The display of LCD text and seven segment display. (b) Configuration of the image memory

10.2.3 Putting the VGA Edge Detection System Together

Based on the discussions so far let us put together an experimental setup edge detector using a DE2-115 board. We would like to have the following features:

- The edge detector should use the 3×3 Sobel mask (10.10) (p. 750).
- The magnitude approximation used the L_1 approximation type; see Table 2.14 (p. 167).
- Images are stored in MIF file format and can be replaced at compile time.
- The filter design takes advantage of the fact that row and column filters are separable, and we use the same row buffer for both filters; see Fig. 10.10b (p. 754).
- The display should be a 640×480 gray scale VGA display at 60 Hz; see Table 10.3 (p. 755)
- We use a circular wrap around of the data.
- The slider switches are used as inputs. The lower 8 bits are used for threshold and the upper 3 bits for different display options.
- The two MSBs of the SW are used of the main image source: 00=Original, 01=Gx, 10=Gy, and 11=|G|.
- The third MSB is used to turn on/off the threshold.
- The seven segment displays are used as follows: 0,1 maximum(Gx), 2,3 maximum(Gy), 4,5 maximum(|G|), and 6,7 frame counter.
- The LCD is used to display the project system information:
 - 1. line: Edge Detection
 - 2. line: DSP with FPGAs

As a starting point design we use the `DE2_115_Default` design from the `DE2_115_demonstration` folder. This includes many of the features we need (actually several more than we need for this project) and many students may be familiar with this since it is the start-up configuration that is the pre-stored factory configuration of the board. Let us now have a look at the

overall system architecture, required control signals, and the Verilog code to run this application.

The VGA chip on the board of the Analog Devices ADV7123KSTZ140 needs `clk` input, `RGB` data, `blank_n`, and `sync_n`. In addition our FPGA circuit needs to provide the horizontal and vertical synchronization signals directly to the VGA monitor. The required 25 MHz clock signal `clk` can be generated by a PLL or just by a division by two of the 50 MHz signal provided to the FPGA. The latter option is small, does not require many resources, and also simplifies simulation. Since we display gray images only, the data are the same for all three channels, i.e., $R=G=B=\text{VGA}$ data. We do not use the `SYNC_N` signal, but we take advantage of the `BLANK_N` signal. In general, when the CRT returns to the beginning of a line or “home” from the end of a frame, the `RGB` signals need to be zero. Alternatively, we can use the `blank_n` signal and tell the VGA monitor when valid data are to be displayed; this simplifies the overall data management and the filter design.

Since we only have 8-bit data and the Sobel filter and magnitude computation may have bit growth, we like to monitor this data. To do so we use the seven segment display and compute continuously the maximum value for G_x , G_y , and the sum $|G_{xy}|$. The two most significant seven segment digits are used to display the frame counter. For a 60 Hz frame rate we expect that the 8-bit counter shows a wrap around after ca. $256/60 \approx 4\text{s}$. Figure 10.11 (p. 755) gives an overview of the overall system design.

The file I/O for the picture is done in the `DE2_115_Default` via the Altera MIF files. The `VGA_DATA` from the `DE2_115` CD contains a small utility program called `PrintNum.exe` that splits any BMP file in a color table and an MIF file that contains the data of the image. BMP format is supported for instance by the MS paint program that comes with MS Windows and under `Image→Attributes` you can scale any image to fit the 640×480 size. Alternatively, you can also write a small MATLAB script that reads an image with the `imread()` function and write it to a MIF file, i.e.:

```
clear;
%% m file to convert 480x640 BMP file into MIF text file
I=imread('BCoin8state.bmp'); %% Read figure
figure, imshow(I); %% Display picture
[r c]=size(I); %% Get number row and columns
rc=r*c; x=1:r*c; J=[];
for k=1:r %% rearrange as 1D array
    J=[J I(k,:)];
end
str=sprintf('Saving %d pixel for picture.mif',rc);
disp(str); str=sprintf('picture.mif'); fid=fopen(str,'w');
fprintf(fid,'depth= %d;\r\nwidth = 8;\r\n',rc);
fprintf(fid,'address_radix = dec;\r\n');
fprintf(fid,'data_radix = dec;\r\n');
```

```

fprintf(fid,'content \r\nbegin\r\n');
for k=1:rc
    fprintf(fid,'%d:%d;\r\n',k-1,J(k));
end
fprintf(fid,'end;\r\n');
fclose('all');

```

In addition to an MIF file we may need to generate some test bench data that can be included in the file. We include this 3×3 test data in the upper left corner of the file. To do that we modify the MIF file data at address 0 – 5, 640 – 645, and 1280 – 1285, i.e.:

```

...
0:10;
1:20;
2:30;
3:0;
4:0;
5:0; ...
640:40;
641:50;
642:60;
643:0;
644:0;
645:0 ; ...
1280:70;
1281:80;
1282:90;
1283:0;
1284:0;
1285:0; ...

```

Now we have enough detail knowledge together to have a look at the implemented Verilog code:

```

module DE2_115_ImageVGA(
// =====
// PORT declarations
// =====
////////// CLOCK //////////
input CLOCK_50,
////////// LED //////////
output [8:0] LEDG,
output [17:0] LEDR,
////////// KEY //////////
input [3:0] KEY,

```

```
////////// SW ///////////
input [17:0] SW,
////////// SEG7 ///////////
output [6:0] HEX0,
output [6:0] HEX1,
output [6:0] HEX2,
output [6:0] HEX3,
output [6:0] HEX4,
output [6:0] HEX5,
output [6:0] HEX6,
output [6:0] HEX7,
////////// LCD ///////////
output LCD_BLON,
inout [7:0] LCD_DATA,
output LCD_EN,
output LCD_ON,
output LCD_RS,
output LCD_RW,
////////// VGA ///////////
output [7:0] VGA_B,
output VGA_BLANK_N,
output VGA_CLK,
output [7:0] VGA_G,
output VGA_HS,
output [7:0] VGA_R,
output VGA_SYNC_N,
output VGA_VS,
//////////Test ports //////
output [10:0] x1_out, x2_out, x3_out,
output [10:0] y1_out, y2_out, y3_out,
output [7:0] tap1_out,
output [7:0] tap2_out);

////////////////////////////
//=====
// REG/WIRE declarations
//=====
parameter Hactive = 640;

reg [7:0] tap1 [1:Hactive];
reg [7:0] tap2 [1:Hactive];
integer I;

wire CPU_CLK;
```

```
wire CPU_RESET;

wire [31:0] mSEG7_DIG;
reg [31:0] Count;

// For VGA Controller
reg VGA_CTRL_CLK;
wire [9:0] mVGA_R;
wire [9:0] mVGA_G;
wire [9:0] mVGA_B;
wire [19:0] mVGA_ADDR;
wire DLY_RST;
wire mVGA_CLK;
wire [9:0] mRed;
wire [9:0] mGreen;
wire [9:0] mBlue;
wire VGA_Read; // VGA data request
wire mDVAL;

//=====
// Data promoted from VGA controller
//=====

wire [7:0] LCD_D_1;
wire LCD_RW_1;
wire LCD_EN_1;
wire LCD_RS_1;
wire iRST_n;
reg oBLANK_n;
reg oHS;
reg oVS;
wire [7:0] b_data;
wire [7:0] g_data;
wire [7:0] r_data;

////// Auxiliary signals
reg [18:0] ADDR;
reg [7:0] gray_data;
wire [10:0] dx, asum, sum, ssum;
wire [8:0] Rxy;
wire [7:0] Gxy;
reg [7:0] Gx, Gy;
reg [10:0] y1, y2, y3;
reg [10:0] x1, x2, x3;
reg [7:0] XY_max, Y_max, X_max;
```

```

wire VGA_CLK_n;
wire [7:0] index;
//reg signed [7:0] gray_data_raw;
wire cBLANK_n, cHS, cVS, rst;
// Display h and v counter values
wire [10:0] h;
wire [9:0] v;
//=====
// Structural coding
//=====

assign VGA_SYNC_N = 1'b0; //not used
//assign VGA_BLANK_N = 1'b1;
assign LCD_DATA = LCD_D_1;
assign LCD_RW = LCD_RW_1;
assign LCD_EN = LCD_EN_1;
assign LCD_RS = LCD_RS_1;
assign LCD_ON = 1'b1;
assign LCD_BLON = 1'b0; //not supported;

always@(posedge oVS or negedge KEY[0])
begin
    if(!KEY[0])
        Count <= 0;
    else
        Count <= Count + 1;
end
assign mSEG7_DIG = { Count[7:0], XY_max, Y_max, X_max};

assign tap1_out = tap1[Hactive];
assign tap2_out = tap2[Hactive];
assign LEDR [15:0] = {v[7:0], h[7:0]};
assign LEDG [7:0] = index; // Memory values

// 7 segment LUT
SEG7_LUT_8 u0 (
    .oSEG0(HEX0), .oSEG1(HEX1), .oSEG2(HEX2), .oSEG3(HEX3),
    .oSEG4(HEX4), .oSEG5(HEX5), .oSEG6(HEX6), .oSEG7(HEX7),
    .iDIG(mSEG7_DIG));

// Reset Delay Timer
Reset_Delay r0(
    .iCLK(VGA_CTRL_CLK),
    .oRESET(DLY_RST));

```

```

// T-FF for 25 MHz signal
always@(posedge CLOCK_50 or negedge KEY[0])
begin
    if(!KEY[0])
        VGA_CTRL_CLK  <=  0;
    else
        VGA_CTRL_CLK  <=  ~VGA_CTRL_CLK;
end

LCD_TEST u5  ( // Host Side
    .iCLK( VGA_CTRL_CLK ),
    .iRST_N(DLY_RST),
    // LCD Side
    .LCD_DATA(LCD_D_1),
    .LCD_RW(LCD_RW_1),
    .LCD_EN(LCD_EN_1),
    .LCD_RS(LCD_RS_1));

// VGA signal generator
assign VGA_CLK = VGA_CTRL_CLK;
assign rst = ~iRST_n;
assign iRST_n = DLY_RST;
assign iVGA_CLK = VGA_CTRL_CLK;
video_sync_generator LTM_ins (
    .vga_clk(iVGA_CLK), // VGA clock at 25 MHz
    .reset(rst), // Reset for
    .blank_n(cBLANK_n), // Data are (not) valid
    .HS(cHS), // Horizontal sync
    .VS(cVS),
    .h(h),.v(v)); // Vertical sync

//// Address generator
always@(posedge iVGA_CLK,negedge iRST_n)
begin
    if (!iRST_n)
        ADDR <= 19'd0;
    else if (cHS==1'b0 && cVS==1'b0)
        ADDR <= 19'd0;
    else if (cBLANK_n==1'b1)
        ADDR <= ADDR + 1;
end

///// Load image data using

```

```

assign VGA_CLK_n = ~VGA_CLK;
img_data img_data_inst (
    .address ( ADDR ),
    .clock ( VGA_CLK_n ),
    .q ( index )
);

////////// Magnitude: max(|Gx|,|Gy|)+ min(|Gx|,|Gy|)/4
assign Rxy = (Gx>Gy) ? {1'h0 ,Gx} + {3'h0 ,Gy[7:2]}
                      : {1'h0 ,Gy} + {3'h0 ,Gx[7:2]};
assign Gxy = (Rxy>8'hFF)? 8'hFF : Rxy[7:0];
assign sum = y1 + 2*y2 + y3; // no register
assign asum = (sum[10]==1'b1) ? -sum : sum;//absolute value
assign ssum = asum/4; // scale appropiate
assign dx =(x1>x3) ? (x1 - x3)/4 : (x3 - x1)/4;
                           // absolute value + difference
assign x1_out = x1;
assign x2_out = x2;
assign x3_out = x3;
assign y1_out = y1;
assign y2_out = y2;
assign y3_out = y3;
always@(posedge iVGA_CLK) ////////// Sobel filter design
begin
    if (cBLANK_n==1'b1) begin
        // Display X, Y, and total gradient maximum values
        if (Gx > X_max) X_max <= Gx;
        if (Gy > Y_max) Y_max <= Gy;
        if (Gxy > XY_max) XY_max <= Gxy;
    //----- abs(Gy) filter -----
    // Horizontal filter [1 2 1]
        if (ssum>8'hFF)
            Gy <= 8'hFF;
        else
            Gy <= ssum[7:0]; // include divide by 4
        y3 <= y2;
        y2 <= y1;           // My vertical filter [1 0 -1] '
        y1 <= {8'h0, index} - {8'h0 ,tap2[Hactive]};
    //----- abs(Gx) Horizontal filter [-1 0 1]
        if (dx>8'hFF) //Threshold
            Gx <= 8'hFF;
        else
            Gx <= dx[7:0];
        x3 <= x2;
    end
end

```

```

x2 <= x1; // Mx vertical filter [1 2 1]
x1 <= {8'h0 ,index} + {8'h0 ,tap1[Hactive],1'b0}
      + {8'h0 ,tap2[Hactive]};

////////// The tap delay lines are used by both Mx and My
for (I=Hactive; I>1; I=I-1) begin
    tap1[I] <= tap1[I-1]; // Tapped delay line 1: shift 1
end
tap1[1] <= index; // Input in register 0
for (I=Hactive; I>1; I=I-1) begin
    tap2[I] <= tap2[I-1]; // Tapped delay line 2: shift 1
end
tap2[1] <= tap1[Hactive]; // Input in register 0
end
end

////////latch valid data at falling edge;
always@(posedge VGA_CLK_n) begin
    case(SW[17:15])
        3'b000 : gray_data <= index;
        3'b001 : begin
            gray_data <= (index < SW[7:0]) ? 8'h0 : 8'hFF; end
        3'b010 : gray_data <= Gx;
        3'b011 : begin
            gray_data <= (Gx < SW[7:0]) ? 8'h0 : 8'hFF; end
        3'b100 : gray_data <= Gy;
        3'b101 : begin
            gray_data <= (Gy < SW[7:0]) ? 8'h0 : 8'hFF; end
        3'b110 : gray_data <= Gxy;
        default : begin
            gray_data <= (Gxy < SW[7:0]) ? 8'h0 : 8'hFF; end
    endcase
end

//////// Delay the iHD, iVD,iDEN for one clock cycle;
always@(negedge VGA_CLK)
begin
    oHS<=cHS;
    oVS<=cVS;
    oBLANK_n<=cBLANK_n;
end

////////// VGA controller output signals
assign VGA_BLANK_N =oBLANK_n;
assign VGA_HS = oHS;

```

```

assign VGA_VS = oVS;
assign VGA_B = gray_data;
assign VGA_G = gray_data;
assign VGA_R = gray_data;

endmodule

```

The Verilog file starts with I/O ports followed by some additional test ports for the simulation only. Then the internal signals follows. The original `vga_controller.v` has been “promoted” to the top level design in order to avoid the many additional ports. The `Structural coding` then assigns first the constant I/O signals. The `always` block for `oVS` has the frame counter that is displayed at segments 6 and 7 of the seven segment display. Next comes the 4-bit to seven segment decoder and the reset delay timer that can be used if there are problem with a short power-up sequence. Then the toggle flip-flop is implemented that divides the 50 MHz input frequency to the 25 MHz clock signals needed for the pixel clock of the VGA display and also serves as a main clock for our Sobel filters. Then the LCD display is instantiated. The TERASIC component allows us to specify an individual text for our application. The file `LCD_TEXT.v` includes the hex coding for the text following the pattern ROM generator for the LCD controller HD44780. We use alphanumeric coding, i.e., 0 starts at 30_{16} , the capital letters at 41_{16} and lower case at 61_{16} . The space is coded as 20_{16} . After the LCD component the VGA synchronization signal component is instantiated. It delivers the horizontal and vertical synchronization signals, row and column counters, and the `cBlank_n` control signals that indicate when new image data are needed. The image is stored in a 1D linear array and whenever the `cBlank_n` we increment the memory counter `ADDR`. The image memory has been designed using the `MegaWizard`. The memory needed is a $2^{19} \times 8$ memory and the MIF file to be used is `VGA_DATA/BCoin8state.mif`. The MIF file for other images can be generated using the `bmp2txt.m` script shown on p. 758. Just make sure that the image is a 640×480 BMP image. Then the actual coding of the Sobel filter starts. The portion of the code that does not need a register is placed outside the `always` statement that is controlled by `iVGA_CLK` and `cBLANK_n`. The first three `if` statements within the `always` are used to determine the maximum for `Gx`, `Gy`, and `Gxy` that is displayed at the seven segment displays. Then the `Gy` filter follows. The vertical filter $[1, 0, -1]^T$ is followed by the horizontal filter $[1, 2, 1]$. This computation together with the absolute value and scaling is placed before the `always` statement. The filter output `Gy` is run through a threshold operation to avoid a wrap around in arithmetic. For `Gx` the first operation is the $[1, 2, 1]^T$ vertical filter followed by the $[1, 0, -1]$ filter. Again scaling and absolute computation can be found before the `always` statement to avoid the need to infer registers. The last part within the `always` statement is the coding for the two length 640 tap long shift registers for the

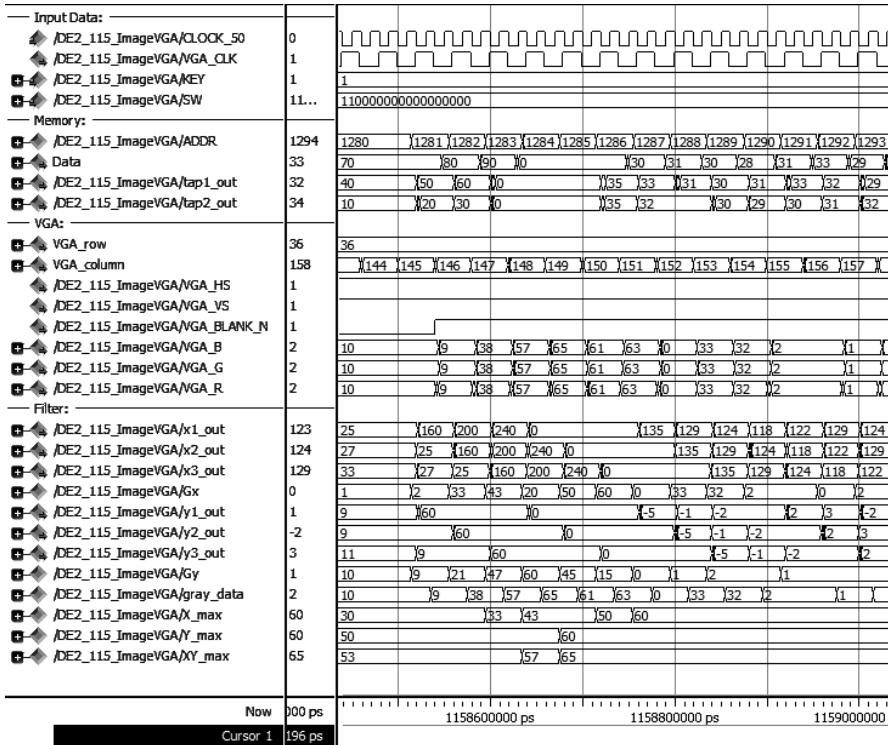


Fig. 10.14. The test bench data processing for the MODELSIM edge detection simulation

row buffer. Here it is important to verify that these are indeed synthesized with block RAM; otherwise a huge number of logic elements would be needed. In particular you *cannot* add a reset to this register file, since there is no reset port in the block RAMs.

Next after the Sobel filter comes the `case` statement that connects the output to the gray level output port. Depending on SW15 this runs through a threshold operation. Four signals can be selected: the original data from memory, the Gx filter output, the Gy filter output, or the magnitude of both named Gxy.

Depending on the precision of the horizontal and vertical synchronization signals in the last `always` the HS, VS, and BLANK_n signals are delayed. Finally in the last set of `assign` statements the selected gray level data is connected to the RGB ports and the three synchronization signals.

Figure 10.12 shown earlier (p. 756) gave a rough overview of the global simulation needed. Simulation time is substantial since the first “useful” data appear after 33 lines have been processed. One line needs 35 μ s or 875 clock cycles. For 33 lines the figure is 28K cycles.

Figure 10.14 shows the details of the test bench after the data for the whole 3×3 for the Sobel mask have arrived. The input data shown are the FPGA 50 MHz clock input and the divide-by-two signal of 25 MHz. The KEY 1 indicates that the reset is not active and comparing the SW= 110... with the case statement in the Verilog code results in the output of Gxy to the VGA monitor, i.e., the magnitude of the edge vector without thresholding is displayed. The next set of data titled **Memory:** shows the image data read from the storage. The ADDR is the address counter, data is the data read from storage, and tap1_out and tap2_out are the outputs of the row buffers that are used for both M_x and M_y Sobel filters. We can now see the test bench data artificially added to the 8 quarter coin state picture as

$$\text{data} = \begin{bmatrix} 70 & 80 & 90 \\ 40 & 50 & 60 \\ 10 & 20 & 30 \end{bmatrix} \quad (10.18)$$

The data are followed by three zero values.

The next block of data entitled **VGA:** shows VGA display related data. It starts with the counters for row and column of the image currently processed. The horizontal and vertical synchronization signals and the **blank_n** signal indicates valid data to display. The RGB signals are the signals sent to the VGA display and all three are equal since we display a gray scale image. The last block of data labeled **Filter:** shows the internal working of the Sobel filter. The **x1** is the row filter for **Gx** that shows the data weighed by $[1, 2, 1]^T$. The computation for first three values are

$$\begin{aligned} x1(0) &= 70 + 2 \times 40 + 10 = 160 \\ x1(1) &= 80 + 2 \times 50 + 20 = 200 \\ x1(2) &= 90 + 2 \times 60 + 30 = 240 \end{aligned}$$

The output **x2** and **x3** then shows the delayed versions need for the horizontal filter $[1, 0, -1]$ and **Gy** the output after filtering, absolute computation, and division by four, e.g.:

$$\begin{aligned} Gx(k) &= |x1(k) - x3(k)| = |240 - 160|/4 = 20 \\ Gx(k+1) &= |x1(k+1) - x3(k+1)| = |200 - 0|/4 = 50 \\ Gx(k+2) &= |x1(k+2) - x3(k+2)| = |240 - 0|/4 = 60 \end{aligned}$$

The scale factor of four is used to avoid an overflow in arithmetic. For the filter **Gy** the **y1** signal shows similar computations. First the row filter $[1, 0, -1]^T$ gives

$$\begin{aligned} y1(0) &= 70 - 10 = 60 \\ y1(1) &= 80 - 20 = 60 \\ y1(2) &= 90 - 30 = 60 \end{aligned}$$

Then the horizontal filter [1, 2, 1] followed by the absolute function and scaling of 4 is computed. We get

$$\begin{aligned}Gy(k) &= |y_1(k) + 2y_2(k) + y_3(k)| = |60 + 2 \times 60 + 60|/4 = 60 \\Gy(k+1) &= |y_1(k+1) + 2y_2(k+1) + y_3(k+1)| \\&= |60 + 2 \times 60 + 0|/4 = 45 \\Gy(k+2) &= |y_1(k+2) + 2y_2(k+2) + y_3(k+2)| = |60|/4 = 15\end{aligned}$$

The output **gray_data**, (i.e., **Gxy**=**VGA_B**) is then computed as magnitude with $\max(|Gx|, |Gy|) + \min(|Gx|, |Gy|)/4$:

$$\begin{aligned}Gxy(k) &= \max(20, 60) + \min(20, 60)/4 = 60 + 20/4 = 65 \\Gxy(k+1) &= \max(50, 45) + \min(50, 45)/4 = 50 + \lfloor 45/4 \rfloor = 61 \\Gxy(k+2) &= \max(60, 15) + \min(60, 15)/4 = 60 + \lfloor 15/4 \rfloor = 63\end{aligned}$$

The last three data show the absolute maximum value so far. Note the increase in the **Gxy** maximum to 65 in this test data.

A final note on the timing simulation. Usually we would have used the functional simulation, but due to the large memory initialization file you will find that your PC will soon run out of memory, i.e., the simulation model for the timing simulation has a more compact memory representation in this particular project.

After we are confident that the simulation works as expected we can generate the programming file, connect a VGA monitor to the DE2-115 board, and download the design to the board. You can then experiment with the different edge gradients in horizontal and vertical directions and also try to find a good threshold for the final binary representation based on the values shown in the seven segment displays.

The synthesis results for the project are shown in the **Compilation Report**. The design uses 816 LEs and no embedded multiplier, and 302 (i.e., 70%) M9K memory blocks. For the slow 85C timing model **TimeQuest** reports **Fmax** = 44.26 MHz registered performance.

10.3 Case Study 2: Median Filter Using an Image Processing Library

Designing an image or video processing system in HDL can be a very time-consuming task since, handling such a large amount of data, the complexity of the system increases rapidly. However, several basic tasks and operations that each image/video processing system has to perform do not differ much and a good set of basic building blocks would be very useful. We discussed such basic operations in Sect. 10.1.2 (p. 746). Altera has two options: there is a professional toolbox that helps for instance with the design of 2D linear and median filters and some image housekeeping operations such as color space

conversion, line buffer compiling and gamma correction to name just a few; see [411]. These blocks can be evaluated, but ultimately require a paid license and the source code and scripts of the blocks are encrypted and cannot be studied or modified. The University program on the other hand supports image and video processing design with a substantial set of basic building blocks, and provides the VHDL and Verilog source code along with the TCL scripts to import the blocks in the SOPC or **Qsys** design environments. A user manual describing how the blocks work and how to instantiate the blocks is available too [412] and is updated on a regular basis as new Quartus II versions become available. We may use these blocks in a stand-alone system and build our own Avalon bus systems. However, most often it is more convenient to use the blocks as co-processor to a Nios II microprocessor system. As discussed in Chap. 9 Altera has announced the discontinuation of the SOPC environment. In the future only the **Qsys** will be supported and we should therefore design our system in **Qsys**. The image processing blocks need to be downloaded from the Altera University program webpage and after installation the blocks can be found under `ip→University_Program→Audio_Video→Video`. Table 10.4 gives a brief overview of the available blocks. All blocks have the Avalon bus interface and the block names therefore start with `altera_up_avalon_video_...`. Details and C-coding examples of how to put the blocks to work can be found in the related documentation entitled “Video IP Cores for Altera DE-Series Boards” [412].

Before we go into details of our system let us have a look at a typical application that fits well for a microprocessor and is in general a challenge for an HDL system.

10.3.1 The Median Filter

In image processing systems *median* filters have proved to be useful in cases when the image processing system is corrupted by “Salt-and-Pepper” (S&P) noise. This may come from defective pixels in a CCD array, too much gain of an intensified digital imager, or data transmission errors [413]. This black-and-white noise can be removed by evaluating the neighborhood of each pixel: if the neighboring pixels are not black-and-white (BW) than it is not likely that the pixel in question should be a BW pixel and we replace the value with the most likely corrected value that is computed as the median of the neighborhood. We do not use the average, i.e., *mean* since mean filters have the tendency to remove details like edges from the image and produce a blurred image. So basically the median filter can be formulated with three easy steps:

- Define the size of a neighborhood such as 5×5 or just horizontal 1×5 or vertical 5×1 that should be used in the median computation.
- Compute the median (i.e., sort and pick the middle value) of a pixel using a sufficiently large neighboring area.

Table 10.4. Image processing blocks in the Altera University Program toolbox

IP	Description
<code>alpha_blender</code>	Combines two video streams
<code>Bayer_resampler</code>	Converts 2×2 Bayer pattern [G1,R;BG2] format to the 24-bit RGB format [R,(G1+G2)/2,B]
<code>Character_buffer_with_dma</code>	Converts ASCII characters into graphical representation for display
<code>chroma_resampler</code>	Converts between YCrCb formats including duplicating, inserting, and dropping pixels
<code>clipper</code>	Modifies the resolution of video stream, i.e., adds or drop lines or columns
<code>csc</code>	Color space converter YCrCb \leftrightarrow RGB
<code>decoder</code>	Reads in video from composite video port or CCD cameras
<code>dma_controller</code>	Stores and retrieves video frames to and from memory. Specifies addressing modes, frame resolution, and pixel format
<code>dual_clock_buffer</code>	Transfer of stream between two clock domains
<code>edge_detection</code>	Edge detection is done in four steps: Gaussian smoothing, Sobel filtering, maximum suppression, and hysteresis
<code>pixel_buffer_dma</code>	Reads video frames from external memory and sets the addressing mode: consecutive or X-Y addressing
<code>rgb_resampler</code>	Allow one to change between different RGB data formats such as 8-, 9-, 16-, 24-, and 30-bit
<code>scaler</code>	Changes width and height resolution by an integer factor
<code>stream_router</code>	Lets you split and merge video streams
<code>test_pattern</code>	Generates a test image in 24-bit RGB. Hue changes from 0 to 360 on the x-axis and saturation changes from 0 to 1 on the y-axis
<code>vga_controller</code>	Generates the synchronization signals for VGA DAC
<code>vip_bridges</code>	Converters video to and from the Altera VIP format

- Replace the pixel with the median value.

The size of the area we should evaluate will depend on the S/N ratio and the computation effort we like to invest. However, keep in mind that an S/N computation is not the best method to characterize the performance of a median filter. An image with little to no S&P noise for instance will have a lower S/N after median filtering since undisturbed pixels will also be replaced with neighboring values. Computing the median seemed reasonably easy, since we just need to sort the data in our window according to their values and then pick from the ordered list the value in the middle. However, sorting is not an easy operation in HW or SW. Before discussing typical HW and SW options let us first have a look at a few example images. For high S/N ratio a small window, say 3×3 , will be sufficient. For low S/N ratios the size of the window becomes substantial larger. Windows of size 9×9 , 11×11 , and 13×13 have been used [413–415]. Also the median filters are obviously not separable; we may still try to use the row/column approach, but should not expect the same performance at the output. In MATLAB we apply a median filter via

```
[I1,m]=imread('TigerGray.bmp');
d=.1
I2 = imnoise(I1,'salt & pepper',d);
I3=medfilt2(I2,[5,5]);
figure,imshow(I3)
```

This script will load an image, add 5% S&P noise to the image, apply a 5×5 median filter, and display the filtered image. Figure 10.15 shows typical filtering examples. Figure 10.15c for the 5×5 mean a.k.a. averaging filter is clearly not a good choice with the S&P noise. The image is blurred rather than cleared of the noise. The 1×5 median followed by a 5×1 filter is competitive with a 5×5 median filter. When we try to remove completely the S&P noise we can use a larger window such as 11×11 shown in Fig. 10.15f. However, notice how the larger filter also may remove some details such as the pattern shown in the wallpaper in the back of the image. These details are better preserved with the shorter filter. A solution to the window size problem may be an adaptive median filter [416].

10.3.2 Median Filter in HDL

Median filtering in HW is a quite challenging task [417] and many attempts have been made to implement efficient sorting methods. FPGA vendors usually supply specialized IP blocks and application notes that cover this type of challenging design [411, 413, 418]. Let us in the following briefly discuss the key ideas.

One of the first proposed methods is the *sorting networks* suggested by Batcher [419]. In each stage two neighboring registered data run through a

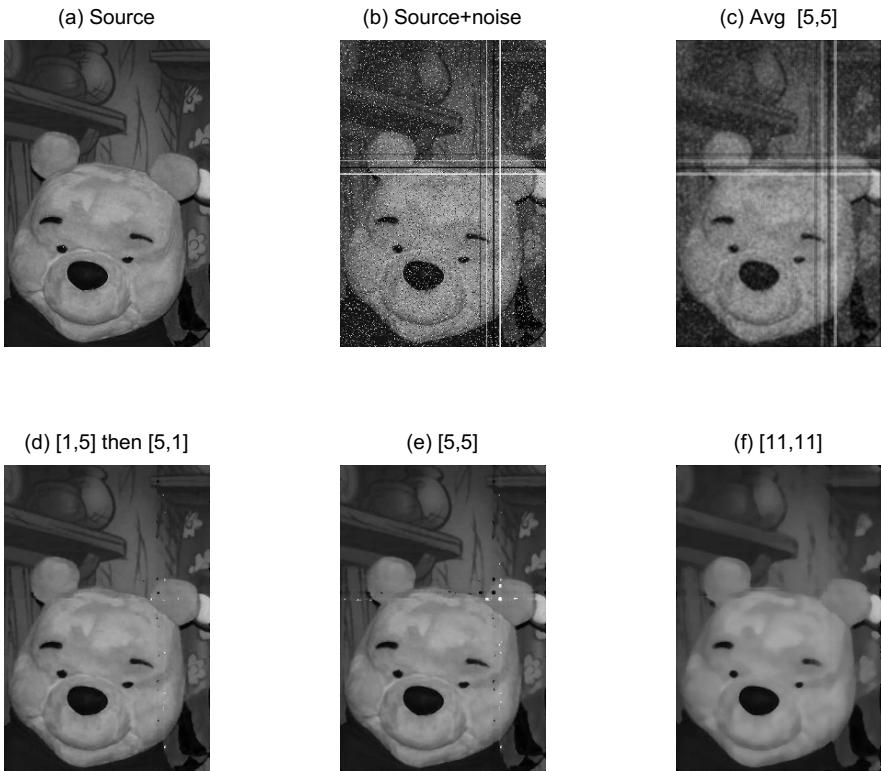
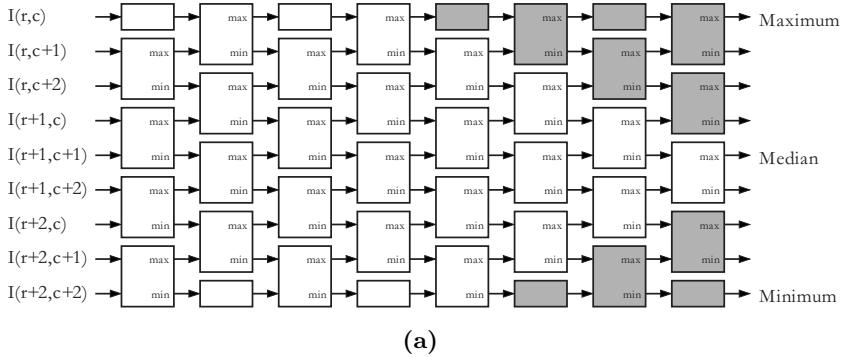


Fig. 10.15. Median and mean filtering examples. (a) The original image. (b) Image disturbed by random, row, and column S&P noise. (c) Mean filtering with size 5×5 . (d) Median filtering by first row filter 1×5 followed by column filtering 5×1 . (e) 5×5 median filtering. (f) 11×11 median filtering

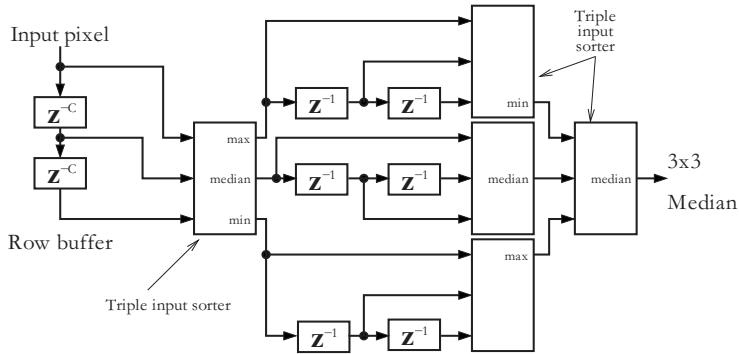
minimum/maximum network. A total of $N(N + 1)/2$ such 2×2 comparisons are needed for 2^N data. Figure 10.16a shows the sorting network for $N = 9$ data. If we implement the $N/2$ comparison elements in parallel the number of steps until sorting is done reduces to $N + 1$.

It is also interesting to note that for the popular 3×3 window it has been shown that a horizontal followed by vertical length 3 median filtering and a final diagonal triple input sorting will produce the correct median value [420], see Fig. 10.16b. However, this scheme cannot be extended to larger median filters. For a 3×3 filter this seemed to be a very attractive design.

A completely different class of median filters works similar to a distributed arithmetic filter and is called *bit voting* [421]. Instead of looking at pairs of data, we now look at a single bit of *all* data at a time. Starting with the MSB the argument of how to find the median is now as follows. Looking at all data count the 1 in the MSBs. Now let us say we have N data to sort and those



(a)



(b)

Fig. 10.16. (a) Batcher sorting for $N = 9$. The small blocks are register and the gray blocks are not required for median computation. **(b)** The row column triple sorter method

we counted are larger than $N/2$. Then we can conclude that the median will be in the set of data that starts with a 1. If we count less than $N/2$ ones (i.e., we have more zeros than ones), then our median will be in the set of data starting with a zero. In the next iteration we only look at the remaining data of interest, i.e., we set a flag to those data that are no longer in the race for the median. Now in the next step we look at the second bit from the MSBs of the remaining data of interest. Again we perform a ones count, and select the value that matches the majority of the second bits of all the data. This continues until the LSB. Figure 10.17 shows the HW architecture of the bit voting method.

The third major group of methods uses a cumulative histogram of the data block [422]. For B bit data we would use 2^B bins. For each incoming

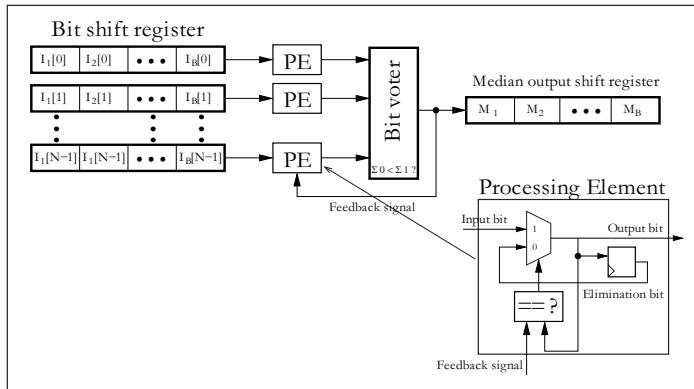


Fig. 10.17. Bit voting architecture using one processing element (PE) per pixel

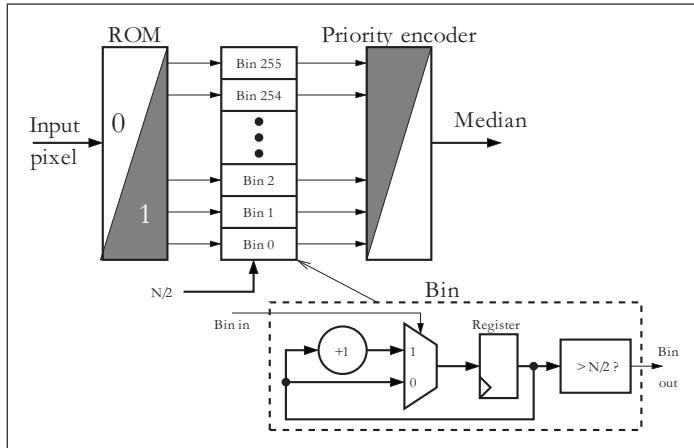


Fig. 10.18. Cumulative histogram architecture

data we increment all bins that belong to the data that are equal or larger than our incoming pixel. At the end we will find the median as the bin with a value of $N/2$. This can be accomplished if we add to each bin a comparator to $N/2$ and a priority decoder that select the smallest bin that is above this threshold. If we like to implement a sliding window median, then we have to subtract the “old” values from our histogram. Figure 10.18 shows the overall architecture.

10.3.3 Nios Median Filtering Image Processing System

When designing a new system we usually have two options. First is starting from scratch and adding more and more components from the **Qsys** library. This approach works well if the system is small and only a few components are

needed. For a large system that uses most of the component on a development board, however, it is usually more time efficient to start with a starting model provided by the board vendor or the FPGA tool vendor. For the DE2 board there are plenty of these starting-point designs available. TERASIC provides several complete systems that you can tailor to your needs. We will use the Altera University program Media Computer VHDL System as a starting-point, since this is available as the Nios **Qsys** version that Altera has announced will be replacing the SOPC builder environment in future software releases [423].

There are several blocks in the Media Processor that are not needed for our purpose, such as **Expansion_JP5**, **PS2_port**, **PS2_Port_Dual**, **Serial_Port**, and **Audio**. The Media Processor uses a reduced resolution of QVGA 320×240 in color but we like to use the whole VGA size of 640×480 to display four gray images at once and therefore also remove the **VGA_Pixel_Scaler** from **Qsys**. Here is a brief listing of the components we keep and the modification necessary to support a 640×480 gray scale image processing:

- **CPU** is the Nios Processor in the standard version with HW support for multiply, 4K instruction cache, and level 2 JTAG debugger; see Chap. 9, p. 694
- **sysid** provides a unique system ID number that simplifies the identification of the USB port the Eclipse finds in the Nios II system
- **merged_resets** provides reset signals for the internal, external, and 27 MHz clocks
- **clk**, **sys_clk**, **vga_clk**, **clk_27** and **External_Clocks** provide the required clock signals for all blocks
- **SDRAM** holds the data and program memory
- **SRAM** is used for the pixel buffer
- **Red_LEDs** as parallel output port to drive the 18 red LEDs
- **Green_LEDs** as parallel output port to drive the 9 green LEDs
- **HEX3_HEX0** and **HEX7_HEX4** each 32-bit output port to drive the eight seven segment displays
- **Slider_switches** as parallel input to 18 switches
- **Pushbuttons** are the four push buttons
- **JTAG_UART** is the Avalon JTAG universal asynchronous receiver transmitter with 64 bytes I/O buffer each
- **Interval_Timer** is the Avalon timer with 32-bit counter size and 125 ms period
- **AV_Config** sets the audio and video configurations
- **VGA_Pixel_Buffer** specifies the X,Y addressing mode, 640×480 resolution, and set the 8-bit gray scale color space used
- **VGA_Pixel_RGB_Resampler** translates the 8-bit gray scale to the 30-bit RGB signal
- **VGA_Char_Buffer** renders the character to be display on the VGA

- **Alpha_Blending** combines the image and character display to a single video stream where the character display is chosen as foreground
- **VGA_Dual_Clock_FIFO** is placed between the alpha blinder and the VGA controller to synchronize the streams
- **VGA_Controller** generates all synchronization signals for the VGA display
- **Char_LCD_16x2** let use display system information on the two row LCD display
- **CPU_fpoin** includes the HW support for faster computation of the four common FP operations: addition, subtraction, multiplication, and division. Note that this block comes from the Nios custom instructions and can be found under `ip→altera→nios2.ip`

The overall system configuration as it appears in **Qsys** is shown in Fig. 10.19. Note that the majority of components came from the Altera University program IP block component library shown on the left. Finally let us compare the saved resources and performance when compared to the original Media Computer. Since the Altera University Program Media Computer also supports audio processing some more resources are needed. From the data shown in Table 10.5 we see the reduced LEs and memory requirement and the improved speed.

Table 10.5. Comparison of media and image processor

Synthesis results	Media computer	Image processor	Improved
F _{max} performance	92.35 MHz	132.17 MHz	43%
LEs	13718	11825	16%
Multiplier	11	11	0%
PLLs	2	1	100%
CPU	9 M9Ks	9 M9Ks	
JTAG	6 M9Ks	2 M9Ks	
VGA	12 M9Ks	10 M9Ks	
Floating-point	15 M9Ks	15 M9Ks	
Audio	4 M9Ks	—	
M9Ks total:	46 M9Ks	36 M9Ks	27%

10.3.4 Median Filter in SW

An HW realtime platform as discussed in the previous section is desirable if you have completed the image processing algorithm evaluation and like to implement it in a realtime video application. As we have seen there is not really a cheap way to build a median filter in HW and substantial development time would be necessary. A median filter that works for different lengths and

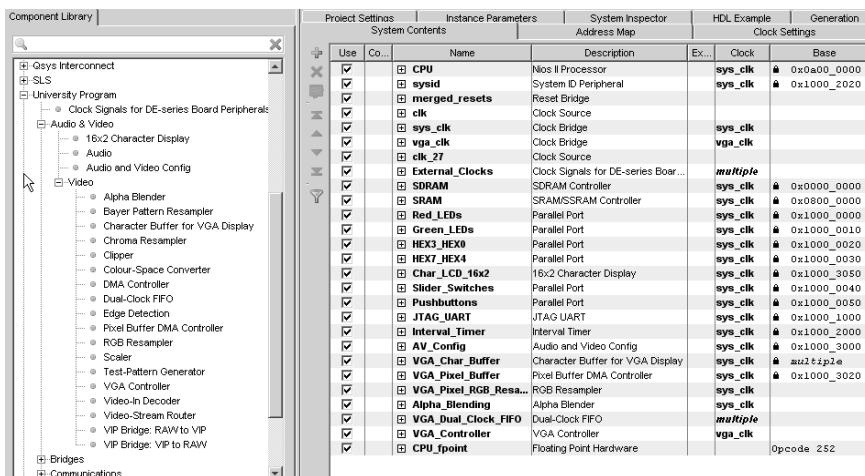


Fig. 10.19. Image processing system configuration

orientations would be an additional challenge. Since we like to experiment with noise levels, filter types, median filter length, and different images, such a stand-alone system would be less flexible. For an experimental setup a median filter in software seemed to be more flexible since in SW we can easily change filter length and orientation.

Before we start with the detailed consideration of how to implement the median filter in software, let us collect some basic facts that relate, to the two development software systems offered by Altera for the Nios II processor: the compact easy to use Altera University Monitor program and the more sophisticated and powerful Nios II Eclipse development system. Images can be stored in Flash, SRAM (via the memory initialization file) or a host. However, the environment to develop an image processing system will benefit from a system that allows the use of an image stored on the host computer. This so-called host-file-system is only available with the Nios II Eclipse running in debug mode. The standard file I/O function such as `fopen`, `fclose`, `fputs`, and `fgets` are available and we therefore need to use the Nios II Eclipse development system.

Since our system has a substantial number of components it is always a good idea to start with a very small SW project such as the “hello world” project and let the Eclipse software generate the required SW driver for the external component. In general we have low level system functions where we can write and read to/from I/O ports like

```
#define switches (volatile char *) 0x01901050
#define      leds (char*)          0x01901060
...
s = *switches;
```

```
printf("New SW value = %d\n",s);
*leds = s;
```

The next higher level of functions is needed when we use any of the more sophisticated components like

```
/* Set the LCD cursor to the home position */
IOWR( LCD_BASE, LCD_WR_COMMAND_REG, 0x02 );
```

that use Altera's High abstraction layer (HAL) functions. The highest abstraction is used if we use ANSI C standard function such as `usleep()` or `fopen()`.

After we have successfully run the “hello world” example we can start with implementing image processing operations. As a goal we like to split our screen into four parts each of size 320×240 pixels such that we can display different image processing operations. Since we have chosen the X-Y addressing mode a pixel index is computed via

```
index = (row << 10) + col;
```

The first routine we need to implement is to open a file on the host system and read in an MIF file in QVGA size. To this end we use the `bmp2txt.m` script from p. 758 and add the line `I=imresize(I,.5);` to scale the 640×480 to a 320×240 image. We will load the picture in the upper left corner of our image memory. Since we are reading an MIF file we only need to check whether the first character, (i.e., ASCII value range between ‘0’ and ‘9’) in a line is a number and then convert the value that appears behind the colon digit by digit. The whole code for the file I/O will look as follows:

```
// -----
printf("File reading begins\n");
pFile=fopen ("/mnt/host/Qpicture.mif","r");
if (pFile==NULL) perror ("Error opening file"); else
{for (;;) {
    for (l = 0; l < 256; l++) Line[l] = ' '; // clear line
    if (fgets(Line, 256, pFile) == 0) break; // file ends
    if ((Line[0]>='0') && (Line[0]<='9')) {
        a=Line[0]; u=0;
        while (a!=':') { u++; a=Line[u];} //Start at :
        u++; a=Line[0]; v=0;
        while (a!=';') { v++; a=Line[v];} //Stop at ;
        d=0;
        for (j=u;j<v;j++) { aa[d]=Line[j]; d++;}
        aa[d]='\0';
        // '0' has 0x30=48_10 ASCII coding
        w=aa[0]-48; // d=1
        if (d>1) w=w*10+aa[1]-48; //d=2
        if (d>2) w=w*10+aa[2]-48; //d=3
```

```

        col = n % 320; row = (int) n / 320;
        pixel_color = w % 256; // all in gray scale
        offset = (row << 10) + col; // compute halfword address
        *(pixel_buffer + offset) = pixel_color; // set pixel
        if ((n%1000)==0) {
            printf("%d Line %s select %s:%d\n",n, Line, aa, w);
        }
        n++;// increment line/pixel counter
    }
}
printf ("\nThe file contains %d pixels.\n",n);
printf("File reading ends\n");
// -----

```

Since the transfer is over the serial JTAG cable the transfer rate is not very high. The transfer of the QVGA image took about 2:32 min. This translates to 43.9 Kbits transfer rate for the HP I7 host computer running MS Windows 7.

In the second step we add “Salt and Pepper” noise to the image according to the slider switch setting. We also add two BW lines in the x and y directions. The code for this is straightforward and can be found on the CD.

In the third step we apply a length- L filter horizontally and the result is displayed in the lower left corner. The core code for this operation is an array sort algorithm. For a microprocessor sorting can be done with merge-sort, heap-sort, quick-sort, histogram based, or the classic linear sort [415, 417]. For short length arrays of less than 20, however, the recommended method is *straight insertion*. For medium length sorts (range 20...50) Shell’s method (a.k.a. diminishing increment sort [86]) that can skip more than one set of data when inserting should a new list element be used. For greater lengths *Quicksort* is recommended [86]. The straight insertion is an order $O(N^2)$ routine and therefore slow for large N , but is compact and can be implemented as shown below:

```

***** sort array *****/
void sort(char *x, int len) {
    int k, j;
    char t;

    for (k = 1; k < len; k++) {
        t = x[j = k + 1];
        while (j != 1 && x[j - 1] > t) {
            x[j] = x[j - 1];
            j--;
        }
        x[j] = t;
    }
}

```

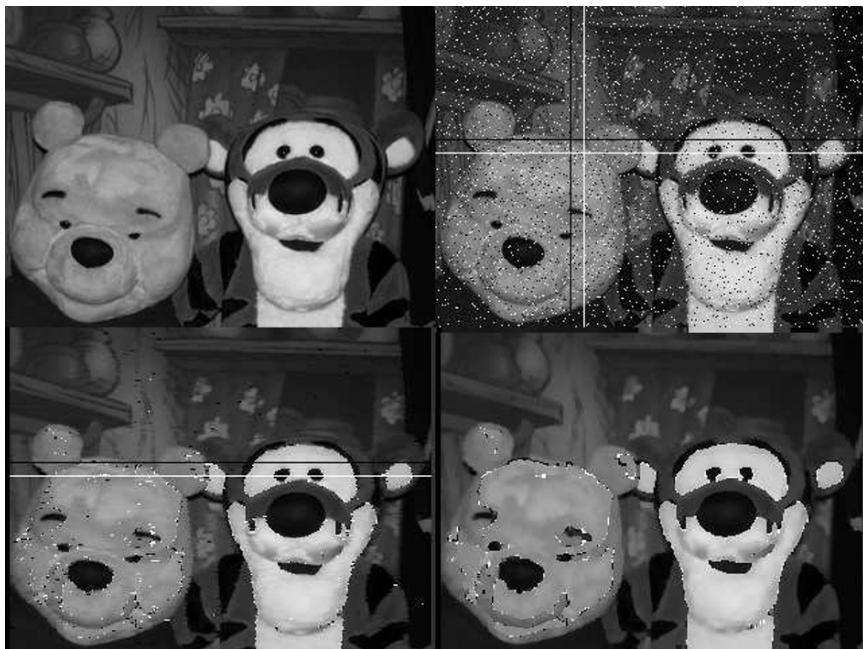


Fig. 10.20. Image processing system result similar to a VGA monitor display

```

    }
}
```

In the final image processing step we apply the vertical filter to the data. The code for this is very similar to horizontal filtering.

We add a “DSP with FPGA” signature to the VGA display. We also store the final QVGA data on our host file system for possible further processing and documentation. We can for instance load the file into MATLAB, reshape the array, and then display as image, i.e.,

```

fid=fopen('results.txt','r');      %% Open file to read
A = fscanf(fid,'%d',[1,inf]);    %% Read figure
I=reshape(A,[480,640]);          %% Put in row/column format
figure, imshow(I/255)            %% Display figure
```

Figure 10.20 shows the host file that was transferred back to the PC and displayed in MATLAB. A length 5 median filter has been used first in horizontal and then in vertical directions. The SW on the board allows one to set the noise level that is added to the image.

10.4 Case Study 3: Motion Detection in Video Processing Improved by Custom Co-processor

Video processing has for a long time been dominated by analog signal processing such as NTSC, PAL, or SECAM systems due to the high bandwidth and data requirements. Images were typically processed with 25–30 frames per second (FPS) that had an analog bandwidth of 6–8 MHz. The even and odd lines were scanned alternately, thereby basically giving the eye the illusion of twice the frame rate.

With the progress in VLSI the digital processing and storing of video frames have now become a reality, as is documented in the various standards from ITU and MPEG. The most research effort in video processing has been spent on the compression of digital video. The amount of data we need to process is enormous. The smallest video size in use is the QCIF format with 176×144 pixels at 30 FPS with 4:2:0 color subsampling and it will already produce 1.1 MB per second raw data. If we look at full HDTV with 1920×1080 at 60 FPS the raw rate goes up to 124 MB/s. A 1-h movie has a raw 446 GBytes data. Substantial data compression is required to fit such a movie onto a DVD or to transmit it in a reasonable time over the Internet [424].

The need to process these large amounts of data quickly makes it difficult to use the latest and most advanced compression techniques such as arithmetic coding and we need to use faster but maybe less effective compression. On the plus side video data include a large amount of redundancy. Think for instance of two neighboring frames in a video sequence. Most of the data will remain the same and only a few pixels may change or move in a certain direction. This can be used in a prediction scheme. Given a current frame at time t (i.e., single image) compute the difference to the next frame at $t + 1$ and transmit only the difference. We can do this directly with the frame in the original domain as we did in the ADPCM speech coding in Chap. 8, or we can use first a transformation such as the DCT and then compute the prediction. Figure 10.21 shows the basic architecture of such a hybrid coding system in the temporal domain.

The favorite transform for image processing is again the DCT since it is close to the optimal KLT and the 2D version can be implemented in a row/column scheme. We have discussed the DCT earlier when we introduced the JPEG standard and we will focus on the implementing of the prediction in this section.

10.4.1 Motion Detection

The most time-consuming operation in the scheme of Fig. 10.21 is the computation of the motion vectors between two adjacent frames. We need to pick a window of reasonable size and then compare within a displacement of $\pm d$ pixels how the block has been moved from one frame to the next. As cost

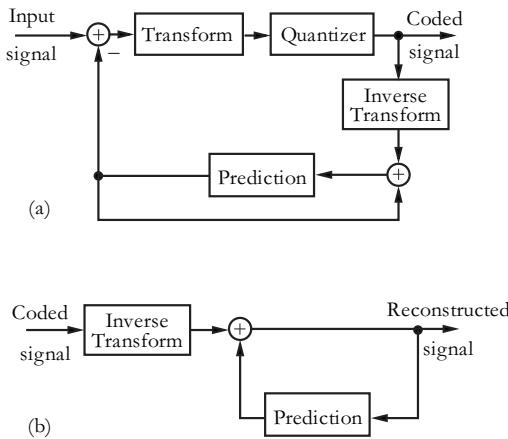


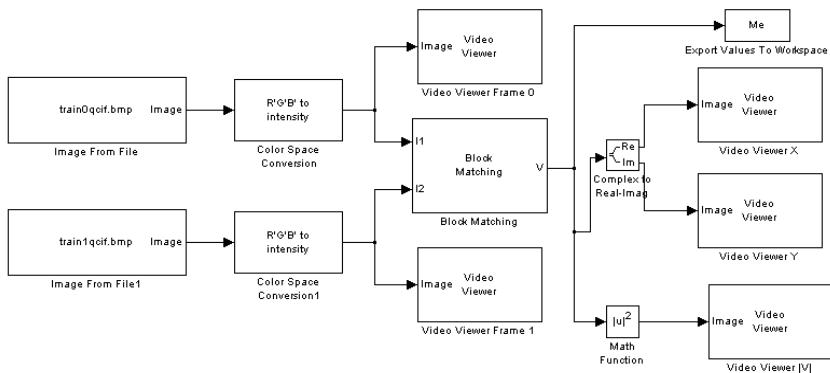
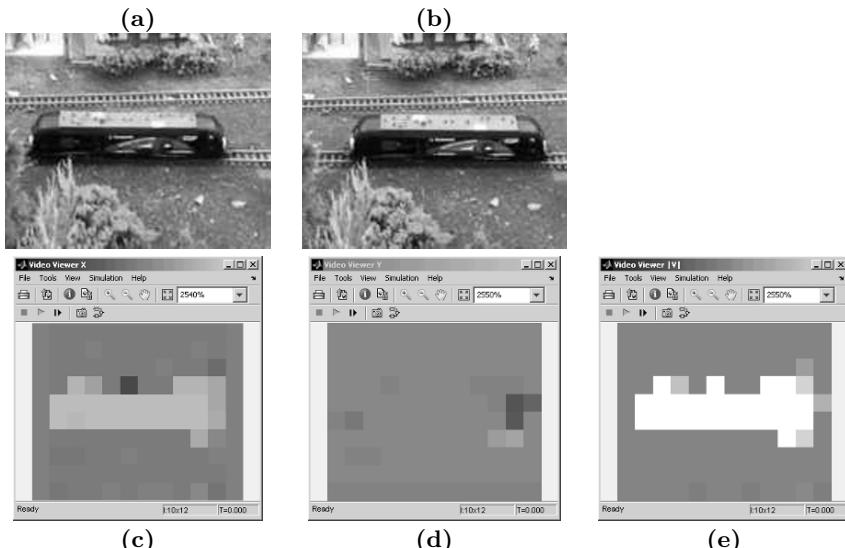
Fig. 10.21. Coding with prediction in the temporal domain

function we may use mean square error (MSE), cross correlation function, or the mean absolute difference (MAD) error. Since the difference between this measure is not that large typically the simplest to implement is used, which is the MAD:

$$MAD(i, j) = \sum_{x=1}^M \sum_{y=1}^N |I_t(x, y) - I_{t-1}(x + i, y + j)| \quad (10.19)$$

with $-d \leq i, j \leq d$ and typically $M = N = 4, 8$, or 16 and a displacement of $d = 16$. In MATLAB such a function is available in the Computer Vision System toolbox in SIMULINK. Figure 10.22 shows the blocks instantiated in SIMULINK. We load two frames from the Quicktime movie *Train.mov* that are already scaled to QCIF format. Then the MAD computation is done according to (10.19). The motion vector can be displayed with $V = X + jY$ coordinates or as an absolute value $|V|$. The test frames are shown in Fig. 10.23 along with the motion vectors. We set both search range (MATLAB: maximum displacement) and block size to 15×15 . With block size of 15×15 (note MATLAB allows only odd length blocks) we get $\lceil 176/15 \rceil = 12$ in the x and $\lceil 144/15 \rceil = 10$ motion values in the y direction, as shown in Fig. 10.23c–e. We see that the train moving from left to right produces many fewer motion vectors in the y than in the x direction. We can export these vectors to the MATLAB workspace and then take a closer look at the values. The MAD range of values is $-11 \dots 9$ in the y direction and $-14 \dots 15$ for the x range.

Since the full search MAD is a time-consuming task we find in the literature many suggestions to speed up the search for the motion vectors. Table 10.6 gives an overview of the different methods suggested. Some of the methods move from a coarse to a fine grid such as the 2D log or three step methods.

**Fig. 10.22.** MATLAB MAD computation using SIMULINK blocks**Fig. 10.23.** MATLAB simulation results for two frames from the `train.mov` Quicktime movie. (a) First frame of moving train. (b) Second frame. (c) Motion vector in x . (d) Motion vector in y . (e) Motion vector magnitude $|V|$

Others move along the gradient by one or more first in the x and then in the y direction. However, keep in mind that these methods will not always find the correct gradient and as VLSI techniques improve we will see more implementations of the full search algorithm since this is the only algorithm that is guaranteed to find the correct motion vector.

Table 10.6. Comparison of motion estimation methods [206, 425, 426]

Method	Steps max	$d = \pm 7$ $n = \lceil \log_2(d) \rceil$ min...max	Reference
Full search	$(2d + 1)^2$	225 ... 225	[427]
2D logarithmic TDL	$8n + 2$	13 ... 26	[428]
Three/N step TSS	$8n + 1$	25 ... 25	[429]
Modified 2D log.	$6n + 1$	13 ... 19	[430]
One-at-a-time CDS	$2d + 3$	5 ... 17	[431]
Orthogonal search OSA	$4n + 1$	13 ... 13	[432]

10.4.2 ME Co-processor Design

For the computation of motion vectors we may take different implementation approaches. In the previous two case studies we saw that the HDL usually gives the best performance and smallest design but less flexibility for changes in data or algorithm. The most flexibility is achieved when using IP blocks as in the median case study that allows a quick change of data and algorithms via change of IP blocks or configuration at the cost of a greater area and less speed. In this third case study we take a middle approach. We still use the flexibility of a Nios II software design, but this time we like to identify slow parts of the algorithm and improved them with a custom co-processor closely coupled with the Nios II processor that can be used via a custom instruction. We had discussed custom instruction (CI) in Chap. 9. See Sect. 9.5.3, p. 721 for more details on the development and instantiation of custom instructions within the **Qsys** environment.

The C-code that is needed to implement the motion estimation has been published in a previous paper [427, 433] and includes the code for the full search algorithm, the three step algorithm, block copy routine (`do_dma` and `GetBlock`), and the cost calculation routine for the MAD and MSE. The code is also included in the book CD. Since we like to improve the full search algorithm let us have a brief look at what the computing requirements are. It turns out that most of the computation (91% in full search and almost 100% in 2DLOG and three step search [434]) is spent on the cost computations and this is therefore the best candidate for a CI improvement.

There is however a small problem we need to solve first to achieve a substantial improvement. The problem arises from differences in the data types we use for CI and image data. The 8-bit type `char` is used to represent image data, while the CI allows us to use 32 bits. It is therefore desired to compute four absolute difference computations by just one CI call. This can be done if we let our `char` and `integer` data use the same starting address. We can then copy the data in the usual style each byte at a time, and when it comes to use the CI we use the `int` pointer starting at the same address.

To verify that this C-code works we can use the following short test (file `MADtest.c`):

```
int ival[3];
char *cval = (char*) ival;
int i = 0,r;

for(i = 0; i < 12; i++) cval[i]=i; //input char
printf("Char values: ");
for(i = 0; i < 12; i++) printf("%x ", cval[i]);
printf("\n");
printf("int value via pointer: ");
for(i = 0; i < 3; i++) printf("%x ", ival[i]);
printf("\n");
r = ALT_CI_MAD_0(ival[0],ival[1]); // Do via CI
printf("CI: MAD A = %d\n",r);
```

which will produced the following output in the Nios transcript window:

```
Char values: 0 1 2 3 4 5 6 7 8 9 a b
int value via pointer: 3020100 7060504 b0a0908
CI: MAD A = 16
```

The 32-bit integer will now access four `char` data at a time that can be transferred to the custom instruction. The MAD computation then gives $|7 - 3| + |6 - 2| + |5 - 1| + |4 - 0| = 16$. Now the VHDL code to implement the MAD is not too complicated. We split both input words into four 8-bit-size words, compute the difference, and then add up the results. The subtraction and absolute value computation is usually simplified for unsigned image values if we use an IF statement such that we can subtract the larger from the smaller value. The PROCESS VHDL code to do the computation is shown below.

```
PROCESS(ncs_cis0_dataaa, ncs_cis0_datab)
  VARIABLE a, b, s, d : STD_LOGIC_VECTOR(11 DOWNTO 0);
BEGIN
  -- WAIT UNTIL clk ='1';
  s := (OTHERS => '0');
  FOR k IN 0 TO 3 LOOP
    a := X"0" & ncs_cis0_dataaa(8*k+7 DOWNTO 8*k);
    b := X"0" & ncs_cis0_datab(8*k+7 DOWNTO 8*k);
    IF a>b THEN -- compute AD
      d := a - b;
    ELSE
      d := b - a;
    END IF;
    s := s + d;
```

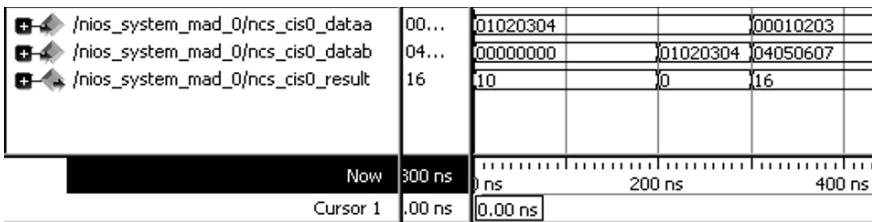


Fig. 10.24. Simulation of the MAD custom instruction using MODELSIM

```

END LOOP;
ncs_cis0_result <= X"00000" & s;
END PROCESS;

```

`ncs_cis0_dataa` and `ncs_cis0_datab` are the input ports and `ncs_cis0_result` is the output port. Such a design only needs a few logic cells (174 LEs) and has an `Fmax` registered performance of over 128 MHz. A simulation of the MAD is shown in Fig. 10.24. Data `a` and `b` are displayed in hex while the `result` is shown in decimal. The last data show for instance the computation of $|7 - 3| + |6 - 2| + |5 - 1| + |4 - 0| = 16$.

Now with the CI working we can run a comparison for the speedup we achieve. We can use the `alt_nticks()` counter and measure the time the software only version and the CI version needs to compute the MAD for a block. For a block of 16×16 we repeat the loop 1000 times to collect enough clock ticks. We will get the following results in the Nios II Console:

```

** Measure the speed first
** Measure the software MAD time
SW 1000 iterations Ticks=32 Time 4000 ms
SW: Cost = 256
** Measure the Altera CI MAD operation
ALT_CI_MAD_0 1000 iterations Ticks=6 Time 750 ms
ALT_CI_MAD_0 speedup = 5
CI: Cost = 256

```

We see that the CI produces a speedup factor of $32/6 = 5.333$. Since this speed measurement is often not used we have put the source code in an extra file called `MADtest.c`.

To run the overall system we need to input at least two frames to the system. There are different approaches we can use:

- Transfer an MIF file as in the first case study.
- Read a video file such as the `foreman.qcif` available from the Internet, e.g., <http://www.cipr.rpi.edu/resource/sequences/> over the host file system.
- Generate a test image.

```

Width: 176 Height: 144 Cur Frame: 1 Ref Frame: 0
Block Size: 16 Algorithm: 1 Search Range: 8 Metric: SAD
** read/generate frames
33 43 62 29 0 8 52 56 56 19
0 0 0 0 0 0 0 0 0 0
Reading frames took 9125 ms
MV Center: x: 0 y: 0
MV Range: X: 0 ... 8 Y: 0 ... 8
MB num: 0 MVx: 2 MVy: 1 MinCost: 256
MV Center: x: 16 y: 0
MV Range: X: 8 ... 24 Y: 0 ... 8
MB num: 1 MVx: 2 MVy: 1 MinCost: 256

```



```

MV Center: x: 144 y: 128
MV Range: X: 136 ... 152 Y: 120 ... 128
MB num: 98 MVx: 7 MVy: -2 MinCost: 7712
MV Center: x: 160 y: 128
MV Range: X: 152 ... 160 Y: 120 ... 128
MB num: 99 MVx: -4 MVy: 0 MinCost: 7476
Total Cost: 166543
Total time: 83875 ms
Program done ...

```

(a)

(b)

Fig. 10.25. Nios Eclipse monitor test. (a) Start of ME measurements (b) End of ME measurements

The generated test image had the distinct advantage that we can include a fixed amount of motion between the two frames we use. The following code segment shows how to generate two frames including motion:

```

//input char
for(i=0; i<nWidth*nHeight; i++) pCur[i]=abs(rand()%100);
for(i=0; i<nWidth+2; i++) pRef[i]=0; //row zeros
for(i=nWidth+2; i<nWidth*nHeight; i++)
    pRef[i]=pCur[i-2-nWidth]+1; //offset

```

The first `for` loop generates the current frame. The second loop adds a zero line for the reference frame, while the last loop copies the frame with a delay of two pixels in x and one line y delay. The motion vector in the run should then show (except for border blocks) for all macro blocks as $x = 2$ and $y = 1$ motion vector. These three `for` loops can easily be modified to implement additional delays in the x or y direction.

Now we have all routines together to run the whole motion estimation. The code for the full search and block copy was published in [427] and can be found on the CD. The simulation results are shown in Fig. 10.25. We use a block size of 16 and a ± 8 pixel search range.

With a block size of 16 we get $[176/16] = 11$ blocks in the x direction and $[144/16] = 9$ in the y directions, or a total of $11 \times 9 = 99$ motion vectors. Note that at the border the correct vector is not always found, particularly at the end of the test, when the y motion vector can only be negative; see Fig. 10.25b.

10.4.3 Video Compression Standards

The development of digital video compression techniques started in the 1980s, when the International Telecommunication Union (ITU) developed a number of video coding standards for real-time transmission applications, such as video conferencing. In video coding we find many fewer non-standard formats

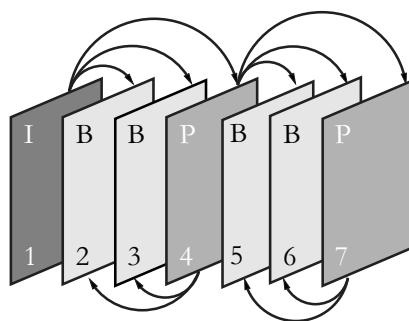


Fig. 10.26. Group of picture coding in MPEG and H.26x

as in still images where many formats are not standardized. The first major aim was H.261 designed for transmission over ISDN lines with data rates multiples of the ISDN data rate of 64 kbytes/s. The major processing steps of the H.261 standard were shown in Fig. 10.21 (p. 783) for differential frame encoding. For intra frame (I-frame) coding a JPEG-like encoding is used (Fig. 10.1, p. 743). Differential inter frame or predictive frame (P-frame) use a coding as in Fig. 10.21, p. 783. In MPEG more frequently bidirectional or B-frames are used that use forward and backward prediction; see Fig. 10.26. The quantization tables in H.261 are not predefined and have to be included in the video stream. A loop filter (1/4, 1/2, 1/4 for H.261) is sometimes used to reduce the blocking effects. Later ITU developed more standards in the H.26x family, such as H.263, H.264, and H.265.

On the other hand ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission) formed MPEG (Moving Picture Experts Group) to develop standards for audio and video compression and transmission such as MPEG-1, MPEG-2, and MPEG-4, 5, and 6. While the H.261 standard was mainly concerned with video compression the MPEG standards also included substantial audio compression effort. Most noticeable is the MPEG-1 layer 3, nowadays briefly call MP3 format, that is the most popular digital audio compression standard in use today.

MPEG-1 aims to meet low complexity requirements, while MPEG-2 was developed for broadcast-quality television and is the most successful video standard today. MPEG-4 is designed especially for low bitrate applications. Joint Video Team (JVT) formed in 2001 by ITU-T Video Coding Experts Group (VCEG) and ISO/IEC MPEG started a new video coding standard, H.264/AVC7,8 completed in 2003. The commonly known MPEG-4 Part 10 is the standard H.264/AVC (Advanced Video Coding), which provides good video quality with lower bitrate than previous coding standards, increasing notably the design complexity. The newest addition is the High Efficiency Video Coding (HEVC) standard, i.e., the ITU-T recommendation H.265 and

MPEG-H, that promises significant improved compression relative to existing standards.

The motion estimation plays a very important role in these video coding standards, widely adopted in MPEG-n and H.26x. ($x = 1 \dots 5$). To simplify the motion estimation, typically each frame is divided into macro blocks (MBs) with fixed sizes. The goal of motion estimation is removing the temporal redundancy between adjacent frames by finding the motion vector (MV) which points up to the best prediction macro block according to our MAD metric (10.19) in the reference frame. The precision used in the MV has been improved from 1 in H.261 to $1/4$ in H.265. The DCT size typically is 8×8 , but H.265 also allows 4×4 , 16×16 , and 32×32 . The macro block size in the past has been 16×16 , but modern standards have a wider choice in macroblock size; see Table 10.7.

The overall processing of the frames is usually done as follows. The first frame, also called intra frame, or I-frame, is coded using standard image compression methods, i.e., DCT, quantization, zig-zag, and VLC coding. Then we may have a series of inter or predicted frames called P-frames. If a coding delay is possible as in MPEG that is target for video distribution and not video conferencing then we may also use a bidirectional frame called B-frame since a prediction is made forward in time from the I-frames and backward from the P-frame. B-frames are usually not (or in restricted form) used in interactive communication such as H.261 or H.263.

Table 10.7. Key parameters of ITU and MPEG video compression standards (MV = motion vector; MB = macroblock) [333, 435–440]

Standard	MV precision	MV range	MB size	Prediction
H.261	1	± 15	16×16	P
MPEG-1	$1/2$	± 1024	16×16	P,B
H.262/MPEG-2	$1/2$	± 2048	$16 \times 16, 16 \times 8$	P,B
H.263	$1/2$	± 256	$16 \times 16, 8 \times 8$	P,B
MPEG-4	$1/4$	± 2048	$16 \times 16, 8 \times 8$	P,B
H.264/AVC	$1/4$	± 2048	$4, 8, 16 \times 4, 8, 16$	P,B
H.265/MPEG-H/HEVC	$1/4$	± 8192	$8 \times 8 \dots 64 \times 64$	P,B

In the earlier coding standards such as H.261 and MPEG-1 we worked with one reference frame; H.264/AVC supports multiple reference frames. This process analyzes the blocks of reference frames in order to estimate the closest block to the current one. Motion vector is therefore an offset from the coordinate of the current MB to the corresponding MB in the reference frame.

Exercises

Note: If you have no prior experience with the Quartus II software, refer to the case study found in Sect. 1.4.3, p. 32. If not otherwise noted use the EP4CE115F29C7 from the Cyclone IV E family for the Quartus II synthesis evaluations. Addition images are available in the folder DE2_115_ImageVGA → VGA_DATA on the CD.

- 10.1:** Use one of your personal pictures or an image from the CD and generate the image file formats (use for instance MS Paint, MATLAB, or HyperSnap from www.hyperionics.com) for BMP, PNG, GIF, and JPEG (Q10).
- Determine the file size, compression ratio (compared to BMP), and image quality.
 - Is compression used? If so, is the compression lossy or lossless?

- 10.2:** Repeat exercise 10.1(a) for the following color maps:

- Monochrome.
- 256 color bitmap.
- 24-bit bitmap.

- 10.3:** Given the image I as an 8×8 matrix shown in Fig. 10.27a:

- Determine J the `dct2` of the image.
- Quantize the 2D DCT using the JPEG quantization matrix from Fig. 10.2a, p. 744.
- Reconstruct the image \hat{I}_r by computing the `idct2` and multiply with the JPEG quantization matrix.
- Determine the average reconstruction error: $1/64 \sum |I - \hat{I}_r|$.
- Use the MATLAB pseudo color function `pcolor()` to plot the four images results from (a)–(d). Are the results for the JPEG coding satisfactory; if not explain why.

- 10.4:** Repeat Exercise 10.3 for the following 8×8 images:

- The Hadamard(8) matrix (use 0 for -1)
- Gray scale image that can be generated with the following MATLAB code:

```
for j=1:8
    for i=1:8
        I(i,j)=10*i+j;
    end
end
```

- Random integer numbers in the range 0...100 using `rng('default');` `I=randi(100,8);`

- 10.5:** After `dct2` transform and quantization with the JPEG matrix shown in Fig. 10.2a, p. 744 the 8×8 matrix $I(r, c)$ with $1 \leq r, c \leq 8$ only has three remaining coefficients: $I(1, 1) = 16$, $I(4, 5) = 1$, and $I(6, 5) = 1$.

- Use the scheme from (10.5) and (10.6), p. 744 to compute the JPEG coding.
- Reconstruct the image \hat{I}_r by computing the `idct2` and multiply with the JPEG quantization matrix.
- Compute the compression gain assuming the original image was coded with 8 bits per pixel.
- Determine the average reconstruction error: $1/64 \sum |I - \hat{I}_r|$ assuming I is `magic(8)` matrix.

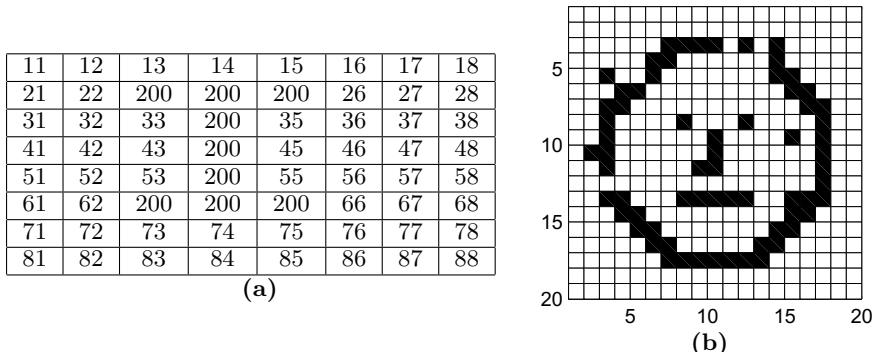


Fig. 10.27. Image data for (a) Exercise 10.3. (b) Exercise 10.14 and Exercise 10.15

10.6: Given a 16×16 pixel gray scale image $I(x, y)$ with

$$I(x, y) = \begin{cases} 100 & x, y \leq 7 \\ 200 & \text{otherwise} \end{cases}$$

with $0 \leq x, y \leq 15$, compute the output image if the image is filtered with:

- (a) Prewitt mask $G_x = I \star M_x$ from (10.9), p. 750.
- (b) Prewitt mask $G_y = I \star M_y$ from (10.9), p. 750.
- (c) Prewitt gradient $|G| = \sqrt{|G_x|^2 + |G_y|^2}$.

10.7: Repeat exercise 10.6 for the Sobel operator from (10.10), p. 750.

10.8: Determine whether the following filter mask can be separated into a row/column filter by determining the matrix rank. For rank one determine the row and column filters and verify that the matrix is generated via the outer product:

(a)

$$M_{3x3} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad (10.20)$$

(b)

$$M_{3x3} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \quad (10.21)$$

(c)

$$M_{5x5}(x, y) = x + y \quad 0 \leq x, y \leq 4 \quad (10.22)$$

10.9: Use one of your personal pictures or a picture from the CD and perform γ correction according to (10.7), p. 746 for:

- (a) $\gamma = 10$.
- (b) $\gamma = 1$.
- (c) $\gamma = 0.2$.

10.10: Use one of your personal pictures or a picture from the CD and reproduce the six images from Fig. 10.15, p. 773. Use MATLAB or C-code to compute the output images:

- (a) Display the original image.
- (b) Image disturbed by random, 1 and 2 rows, and column S&P noise.
- (c) Mean filtering with size 5×5 .
- (d) Median filtering by first row filter 1×5 followed by column filtering 5×1 .
- (e) 5×5 median filtering.
- (f) 11×11 median filtering.

10.11: (a) Load and display the file `Coin5Quarter.bmp` from the CD.

- (b) Compute and display the three modal pixel histogram. Determine the three maxima.
- (c) Find a threshold that separates the maxima one and two.
- (d) Find a threshold that separates maxima two and three.
- (e) Which threshold let you read the text on the coins more easily?

10.12: Given is the following eight value gray scale histogram

Gray scale	0	40	80	100	120	140	180	220
# pixels	1000	500	3000	5000	6000	4000	100	200

- (a) How many pixels has the image?
- (b) Plot the histogram for the image.
- (c) Apply the histogram equalization to the image using the following rule:

$$G_{\text{new}} = \frac{\text{Number of pixels with gray scale } \leq G_{\text{old}}}{\text{Number of all pixels}} \text{ Maximum gray scale level.}$$

What are the new gray scale values of the image?

- (d) Draw the histogram of the transformed image.

10.13: (a) Use one of your personal pictures or an image from the CD and generate an 8-bit gray scale image (use for instance MS Paint, MATLAB, or HyperSnap from www.hyperionics.com).

- (b) Transform the image via the point operation: $I_{\text{new}} = I/8$. Plot the images and the gray scale pixel histograms before and after the transformation.
- (c) Transform the image via the point operation: $I_{\text{new}} = I/8 + 200$. Plot the images and the gray scale pixel histograms before and after the transformation.
- (d) Apply a histogram equalization transformation T to the I_{new} from (b) such that the cumulative histogram become linear

```
histI = imhist(I,256)';
cum = cumsum(histI);
T=cum/max(cum)*256;
```

Plot T , and the histograms before and after the transformation.

- (e) Repeat (d) for the I_{new} from (c).

10.14: Apply a 2×2 dilation operation to the image shown in Fig. 10.27b. White is 0 and black is 1. To replace the center pixel:

- (a) Use a 3×3 majority black vote.
- (b) Use a 3×3 maximum replacement.
- (c) Compare the results from (a) and (b). Are the operations congruent?

10.15: Apply first a dilation and then a 3×3 erosion operation to the image shown in Fig. 10.27b. White is 0 and black is 1. To replace the center pixel:

- (a) Use a 3×3 majority white vote.
- (b) Use a 3×3 minimum replacement.
- (c) Compare the results to the original image. Are the operations congruent?

10.16: Modify the VGA design from case study 1:

- (a) Include your name in the second line of the LCD display.
- (b) Change the image to one of your personal pictures or another image from the CD using the `bmp2mif.m` script or `PrintNum.exe`. Download the new image and make sure the maximum Gx and Gy values are less than FF, otherwise adjust the scaling within the VHDL code.
- (c) Add simulation test data in the upper left corner of the image

```
10 40 70
20 50 80
30 60 90
```

and perform a simulation similar to Fig. 10.14, p. 767.

10.17: Explain why S/N is not a good measure for median filters. Give an example.

10.18: Use the example `TigerGray.bmp` image and add random S&P noise, and add three lines S&H noise to it. Increase the median filter to 7×7 , 9×9 , and 11×11 . Does the results improve?

10.19: Implement an $N = 9$ Batcher sorter [419] for minimum, maximum, and median in HDL (see Fig. 10.16a, p. 774) and test the circuit with data such as $1, 2, 3, \dots, 9$ and $9, 8, \dots, 1$.

10.20: Implement the bit voting architecture [421] for a 3×3 sorter in HDL (see Fig. 10.17, p. 775) and test it with data such as $1, 2, 3, \dots, 9$ and $9, 8, \dots, 1$.

10.21: (a) Take two frames from the train video, scale them to QCIF, and compute the moving vectors for 8×8 blocks using MATLAB/SIMULINK.

- (b) Repeat the experiment with the Nios Image Processing system from the third case study.

10.22: To test the moving estimation designs use random images and $x = 2$; $y = 1$ moving vectors, QCIF image size, and 8×8 blocks:

- (a) Implement the one-at-a-time moving estimation search in software.
- (b) Implement the one-at-a-time moving estimation search as CI.
- (c) Compare the design in terms of resources and speed to the full search.
- (d) Perform the cost evaluation with/without CI.

Appendix A. Verilog Code of Design Examples

The next pages contain the Verilog 1364-2001 code of all design examples. The old style Verilog 1364-1995 code can be found in [441]. The synthesis results for the examples are listed on page 881.

```
//*****
// IEEE STD 1364-2001 Verilog file: example.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module example //----> Interface
  #(parameter WIDTH =8) // Bit width
  (input clk, // System clock
   input reset, // Asynchronous reset
   input [WIDTH-1:0] a, b, op1, // Vector type inputs
   output [WIDTH-1:0] sum, // Vector type inputs
   output [WIDTH-1:0] c, // Integer output
   output reg [WIDTH-1:0] d); // Integer output
// -----
reg [WIDTH-1:0] s; // Infer FF with always
wire [WIDTH-1:0] op2, op3;
wire [WIDTH-1:0] a_in, b_in;

assign op2 = b; // Only one vector type in Verilog;
               // no conversion int -> logic vector necessary

lib_add_sub add1 //----> Component instantiation
( .result(op3), .dataa(op1), .datab(op2));
  defparam add1.lpm_width = WIDTH;
  defparam add1.lpm_direction = "SIGNED";

lib_ff reg1
( .data(op3), .q(sum), .clock(clk)); // Used ports
  defparam reg1.lpm_width = WIDTH;

assign c = a + b; //----> Data flow style (concurrent)
assign a_i = a; // Order of statement does not
```

```

assign b_i = b; // matter in concurrent code

//----> Behavioral style
always @(posedge clk or posedge reset)
begin : p1           // Infer register
    reg [WIDTH-1:0] s;
    if (reset) begin
        s = 0; d = 0;
    end else begin
        //s <= s + a_i;      // Signal assignment statement
        // d = s;
        s = s + b_i;
        d = s;
    end
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: fun_text.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// A 32 bit function generator using accumulator and ROM
// -----
module fun_text          //----> Interface
#(parameter WIDTH = 32) // Bit width
(input  clk,             // System clock
 input  reset,            // Asynchronous reset
 input  [WIDTH-1:0] M,   // Accumulator increment
 output reg [7:0] sin,    // System sine output
 output reg [7:0] acc);  // Accumulator MSBs
// -----
reg [WIDTH-1:0] acc32;
wire [7:0]     msbs;       // Auxiliary vectors
reg [7:0]     rom[255:0];

always @(posedge clk or posedge reset)
if (reset == 1)
    acc32 <= 0;
else begin
    acc32 <= acc32 + M; //-- Add M to acc32 and
end                         //-- store in register

assign msbs = acc32[WIDTH-1:WIDTH-8];

```

```

assign acc  = msbs;

initial
begin
$readmemh("sine256x8.txt", rom);
end

always @ (posedge clk)
begin
sin <= rom[msbs];
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: cmul7p8.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module cmul7p8          // -----> Interface
  (input  signed [4:0] x,           // System input
   output signed [4:0] y0, y1, y2, y3); // The 4 system outputs y=7*x/8
// -----
  assign y0 = 7 * x / 8;
  assign y1 = x / 8 * 7;
  assign y2 = x/2 + x/4 + x/8;
  assign y3 = x - x/8;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: add1p.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module add1p
#(parameter WIDTH    = 19, // Total bit width
        WIDTH1   = 9, // Bit width of LSBs
        WIDTH2   = 10) // Bit width of MSBs
  (input  [WIDTH-1:0] x, y, // Inputs
   output [WIDTH-1:0] sum, // Result
   input      clk, // System clock
   output      LSBs_carry); // Test port

  reg [WIDTH1-1:0] l1, l2, s1; // LSBs of inputs

```

```

    reg [WIDTH1:0] r1;           // LSBs of inputs
    reg [WIDTH2-1:0] l3, l4, r2, s2; // MSBs of input
// -----
    always @(posedge clk) begin
        // Split in MSBs and LSBs and store in registers
        // Split LSBs from input x,y
        l1[WIDTH1-1:0] <= x[WIDTH1-1:0];
        l2[WIDTH1-1:0] <= y[WIDTH1-1:0];
        // Split MSBs from input x,y
        l3[WIDTH2-1:0] <= x[WIDTH2-1+WIDTH1:WIDTH1];
        l4[WIDTH2-1:0] <= y[WIDTH2-1+WIDTH1:WIDTH1];
    **** First stage of the adder ****
        r1 <= {1'b0, l1} + {1'b0, l2};
        r2 <= l3 + l4;
    **** Second stage of the adder ****
        s1 <= r1[WIDTH1-1:0];
        // Add MSBs (x+y) and carry from LSBs
        s2 <= r1[WIDTH1] + r2;
    end

    assign LSBs_carry = r1[WIDTH1]; // Add a test signal

    // Build a single registered output word
    // of WIDTH = WIDTH1 + WIDTH2
    assign sum = {s2, s1};

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: add2p.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// 22-bit adder with two pipeline stages
// uses no components
module add2p
#(parameter WIDTH    = 28,      // Total bit width
        WIDTH1   = 9,       // Bit width of LSBs
        WIDTH2   = 9,       // Bit width of middle
        WIDTH12  = 18,      // Sum WIDTH1+WIDTH2
        WIDTH3   = 10)     // Bit width of MSBs
(input  [WIDTH-1:0] x, y,    // Inputs
 output [WIDTH-1:0] sum,    // Result
 output LSBs_carry, MSBs_carry, // Carry test bits
 input   clk);             // System clock
// -----

```

```

reg [WIDTH1-1:0] l1, l2, v1, s1; // LSBs of inputs
reg [WIDTH1:0] q1; // LSBs of inputs
reg [WIDTH2-1:0] l3, l4, s2; // Middle bits
reg [WIDTH2:0] q2, v2; // Middle bits
reg [WIDTH3-1:0] l5, l6, q3, v3, s3; // MSBs of input

// Split in MSBs and LSBs and store in registers
always @(posedge clk) begin
    // Split LSBs from input x,y
    l1[WIDTH1-1:0] <= x[WIDTH1-1:0];
    l2[WIDTH1-1:0] <= y[WIDTH1-1:0];
    // Split middle bits from input x,y
    l3[WIDTH2-1:0] <= x[WIDTH2-1+WIDTH1:WIDTH1];
    l4[WIDTH2-1:0] <= y[WIDTH2-1+WIDTH1:WIDTH1];
    // Split MSBs from input x,y
    l5[WIDTH3-1:0] <= x[WIDTH3-1+WIDTH12:WIDTH12];
    l6[WIDTH3-1:0] <= y[WIDTH3-1+WIDTH12:WIDTH12];
    //***** First stage of the adder *****
    q1 <= {1'b0, l1} + {1'b0, l2}; // Add LSBs of x and y
    q2 <= {1'b0, l3} + {1'b0, l4}; // Add LSBs of x and y
    q3 <= l5 + l6; // Add MSBs of x and y
    //***** Second stage of the adder *****
    v1 <= q1[WIDTH1-1:0]; // Save q1
    // Add result from middle bits (x+y) and carry from LSBs
    v2 <= q1[WIDTH1] + {1'b0,q2[WIDTH2-1:0]};
    // Add result from MSBs bits (x+y) and carry from middle
    v3 <= q2[WIDTH2] + q3;
    //***** Third stage of the adder *****
    s1 <= v1; // Save v1
    s2 <= v2[WIDTH2-1:0]; // Save v2
    // Add result from MSBs bits (x+y) and 2. carry from middle
    s3 <= v2[WIDTH2] + v3;
end

assign LSBs_carry = q1[WIDTH1]; // Provide test signals
assign MSBs_carry = v2[WIDTH2];

// Build a single output word of WIDTH=WIDTH1+WIDTH2+WIDTH3
assign sum ={s3, s2, s1}; // Connect sum to output pins

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: add3p.v

```

```

// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// 37-bit adder with three pipeline stage
// uses no components

module add3p
#(parameter WIDTH      = 37, // Total bit width
        WIDTH0       = 9,  // Bit width of LSBs
        WIDTH1       = 9,  // Bit width of 2. LSBs
        WIDTH01      = 18, // Sum WIDTH0+WIDTH1
        WIDTH2       = 9,  // Bit width of 2. MSBs
        WIDTH012     = 27, // Sum WIDTH0+WIDTH1+WIDTH2
        WIDTH3       = 10) // Bit width of MSBs
    (input  [WIDTH-1:0] x, y, // Inputs
     output [WIDTH-1:0] sum, // Result
     output LSBs_Carry, Middle_Carry, MSBs_Carry, // Test pins
     input      clk); // Clock
// -----
    reg [WIDTH0-1:0] 10, 11, r0, v0, s0;      // LSBs of inputs
    reg [WIDTH0:0] q0;                         // LSBs of inputs
    reg [WIDTH1-1:0] 12, 13, r1, s1;          // 2. LSBs of input
    reg [WIDTH1:0] v1, q1;                     // 2. LSBs of input
    reg [WIDTH2-1:0] 14, 15, s2, h7;          // 2. MSBs bits
    reg [WIDTH2:0] q2, v2, r2;                // 2. MSBs bits
    reg [WIDTH3-1:0] 16, 17, q3, v3, r3, s3, h8;
                                            // MSBs of input

always @ (posedge clk) begin
// Split in MSBs and LSBs and store in registers
// Split LSBs from input x,y
10 [WIDTH0-1:0] <= x [WIDTH0-1:0];
11 [WIDTH0-1:0] <= y [WIDTH0-1:0];
// Split 2. LSBs from input x,y
12 [WIDTH1-1:0] <= x [WIDTH1-1+WIDTH0:WIDTH0];
13 [WIDTH1-1:0] <= y [WIDTH1-1+WIDTH0:WIDTH0];
// Split 2. MSBs from input x,y
14 [WIDTH2-1:0] <= x [WIDTH2-1+WIDTH01:WIDTH01];
15 [WIDTH2-1:0] <= y [WIDTH2-1+WIDTH01:WIDTH01];
// Split MSBs from input x,y
16 [WIDTH3-1:0] <= x [WIDTH3-1+WIDTH012:WIDTH012];
17 [WIDTH3-1:0] <= y [WIDTH3-1+WIDTH012:WIDTH012];

//***** First stage of the adder *****
q0 <= {1'b0, 10} + {1'b0, 11}; // Add LSBs of x and y

```

```

q1 <= {1'b0, 12} + {1'b0, 13}; // Add 2. LSBs of x / y
q2 <= {1'b0, 14} + {1'b0, 15}; // Add 2. MSBs of x/y
q3 <= 16 + 17; // Add MSBs of x and y
//***** Second stage of the adder *****
v0 <= q0[WIDTH0-1:0]; // Save q0
// Add result from 2. LSBs (x+y) and carry from LSBs
v1 <= q0[WIDTH0] + {1'b0, q1[WIDTH1-1:0]};
// Add result from 2. MSBs (x+y) and carry from 2. LSBs
v2 <= q1[WIDTH1] + {1'b0, q2[WIDTH2-1:0]};
// Add result from MSBs (x+y) and carry from 2. MSBs
v3 <= q2[WIDTH2] + q3;

//***** Third stage of the adder *****
r0 <= v0; // Delay for LSBs
r1 <= v1[WIDTH1-1:0]; // Delay for 2. LSBs
// Add result from 2. MSBs (x+y) and carry from 2. LSBs
r2 <= v1[WIDTH1] + {1'b0, v2[WIDTH2-1:0]};
// Add result from MSBs (x+y) and carry from 2. MSBs
r3 <= v2[WIDTH2] + v3;
//***** Fourth stage of the adder *****
s0 <= r0; // Delay for LSBs
s1 <= r1; // Delay for 2. LSBs
s2 <= r2[WIDTH2-1:0]; // Delay for 2. MSBs
// Add result from MSBs (x+y) and carry from 2. MSBs
s3 <= r2[WIDTH2] + r3;
end

assign LSBs_Carry = q0[WIDTH1]; // Provide test signals
assign Middle_Carry = v1[WIDTH1];
assign MSBs_Carry = r2[WIDTH2];

// Build a single output word of
// WIDTH = WIDTH0 + WIDTH1 + WIDTH2 + WIDTH3
assign sum = {s3, s2, s1, s0}; // Connect sum to output

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: div_res.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// Restoring Division
// Bit width: WN WD WN WD
// Nominator / Denumerator = Quotient and Remainder

```

```

// OR:      Nominator = Quotient * Denumerator + Remainder
// -----
module div_res          //-----> Interface
  (input clk,           // System clock
   input reset,         // Asynchron reset
   input [7:0] n_in,    // Nominator
   input [5:0] d_in,    // Denumerator
   output reg [5:0] r_out, // Remainder
   output reg [7:0] q_out); // Quotient
// -----
reg [1:0] state;        // FSM state
parameter ini=0, sub=1, restore=2, done=3; // State
                                         // assignments
// Divider in behavioral style
always @(posedge clk or posedge reset)
begin : States // Finite state machine
  reg [3:0] count;

  reg [13:0] d;        // Double bit width unsigned
  reg signed [13:0] r; // Double bit width signed
  reg [7:0] q;

  if (reset) begin      // Asynchronous reset
    state <= ini; count <= 0;
    q <= 0; r <= 0; d <= 0; q_out <= 0; r_out <= 0;
  end else
    case (state)
      ini : begin        // Initialization step
        state <= sub;
        count = 0;
        q <= 0;           // Reset quotient register
        d <= d_in << 7; // Load aligned denumerator
        r <= n_in;        // Remainder = nominator
      end
      sub : begin        // Processing step
        r <= r - d;      // Subtract denumerator
        state <= restore;
      end
      restore : begin    // Restoring step
        if (r < 0) begin // Check r < 0
          r <= r + d;    // Restore previous remainder
          q <= q << 1;   // LSB = 0 and SLL
        end
      end
    endcase
  end
end

```

```

        q <= (q << 1) + 1; // LSB = 1 and SLL
        count = count + 1;
        d <= d >> 1;

        if (count == 8)    // Division ready ?
            state <= done;
        else
            state <= sub;
    end
    done : begin      // Output of result
        q_out <= q[7:0];
        r_out <= r[5:0];
        state <= ini;   // Start next division
    end
endcase
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: div_aegp.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// Convergence division after
// Anderson, Earle, Goldschmidt, and Powers
// Bit width: WN      WD      WN      WD
//             Nominator / Denumerator = Quotient and Remainder
// OR:       Nominator = Quotient * Denumerator + Remainder
// -----
module div_aegp
    (input clk,           // System clock
     input reset,         // Asynchron reset
     input [8:0] n_in,    // Nominator
     input [8:0] d_in,    // Denumerator
     output reg [8:0] q_out); // Quotient
// -----
    reg [1:0] state;
    always @ (posedge clk or posedge reset) // -> Divider in
    begin : States                      // behavioral style
        parameter s0=0, s1=1, s2=2;
        reg [1:0] count;

        reg [9:0] x, t, f;           // one guard bit
        reg [17:0] tempx, tempt;

```

```

if (reset) begin          // Asynchronous reset
    state <= s0; q_out <= 0; count = 0; x <= 0; t <= 0;
end else
case (state)
    s0 : begin          // Initialization step
        state <= s1;
        count = 0;
        t <= {1'b0, d_in}; // Load denominator
        x <= {1'b0, n_in}; // Load nominator
    end
    s1 : begin          // Processing step
        f = 512 - t;      // TWO - t
        tempx = (x * f); // Product in full
        tempt = (t * f); // bitwidth
        x <= tempx >> 8; // Fractional f
        t <= tempt >> 8; // Scale by 256
        count = count + 1;
        if (count == 2)    // Division ready ?
            state <= s2;
        else
            state <= s1;
    end
    s2 : begin          // Output of result
        q_out <= x[8:0];
        state <= s0;    // Start next division
    end
endcase
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: cordic.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module cordic #(parameter W = 7) // Bit width - 1
(input clk,           // System clock
 input reset,         // Asynchronous reset
 input signed [W:0] x_in, // System real or x input
 input signed [W:0] y_in, // System imaginary or y input
 output reg signed [W:0] r, // Radius result
 output reg signed [W:0] phi, // Phase result
 output reg signed [W:0] eps); // Error of results
// -----

```

```

// There is bit access in Quartus array types
// in Verilog 2001, therefore use single vectors
// but use a separate lines for each array!
reg signed [W:0] x [0:3];
reg signed [W:0] y [0:3];
reg signed [W:0] z [0:3];

always @ (posedge reset or posedge clk) begin : P1
    integer k; // Loop variable
    if (reset) begin // Asynchronous clear
        for (k=0; k<=3; k=k+1) begin
            x[k] <= 0; y[k] <= 0; z[k] <= 0;
        end
        r <= 0; eps <= 0; phi <= 0;
    end else begin
        if (x_in >= 0) // Test for x_in < 0 rotate
            begin // 0, +90, or -90 degrees
                x[0] <= x_in; // Input in register 0
                y[0] <= y_in;
                z[0] <= 0;
            end
        else if (y_in >= 0)
            begin
                x[0] <= y_in;
                y[0] <= - x_in;
                z[0] <= 90;
            end
        else
            begin
                x[0] <= - y_in;
                y[0] <= x_in;
                z[0] <= -90;
            end
    end

    if (y[0] >= 0) // Rotate 45 degrees
        begin
            x[1] <= x[0] + y[0];
            y[1] <= y[0] - x[0];
            z[1] <= z[0] + 45;
        end
    else
        begin
            x[1] <= x[0] - y[0];
        end

```

```

        y[1] <= y[0] + x[0];
        z[1] <= z[0] - 45;
    end

    if (y[1] >= 0)                                // Rotate 26 degrees
        begin
            x[2] <= x[1] + (y[1] >>> 1); // i.e. x[1]+y[1]/2
            y[2] <= y[1] - (x[1] >>> 1); // i.e. y[1]-x[1]/2
            z[2] <= z[1] + 26;
        end
    else
        begin
            x[2] <= x[1] - (y[1] >>> 1); // i.e. x[1]-y[1]/2
            y[2] <= y[1] + (x[1] >>> 1); // i.e. y[1]+x[1]/2
            z[2] <= z[1] - 26;
        end

    if (y[2] >= 0)                                // Rotate 14 degrees
        begin
            x[3] <= x[2] + (y[2] >>> 2); // i.e. x[2]+y[2]/4
            y[3] <= y[2] - (x[2] >>> 2); // i.e. y[2]-x[2]/4
            z[3] <= z[2] + 14;
        end
    else
        begin
            x[3] <= x[2] - (y[2] >>> 2); // i.e. x[2]-y[2]/4
            y[3] <= y[2] + (x[2] >>> 2); // i.e. y[2]+x[2]/4
            z[3] <= z[2] - 14;
        end

        r    <= x[3];
        phi <= z[3];
        eps <= y[3];
    end
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: arctan.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
// -----
module arctan #(parameter W = 9,      // Bit width
                  L = 5)       // Array size
    (input clk,           // System clock

```

```

input reset,           // Asynchron reset
input signed [W-1:0] x_in, // System input
//output reg signed [W-1:0] d_o [1:L],
output wire signed [W-1:0] d_o1, d_o2 ,d_o3, d_o4 ,d_o5,
                           // Auxiliary recurrence
output reg signed [W-1:0] f_out); // System output
// -----
reg signed [W-1:0] x; // Auxilary signals
wire signed [W-1:0] f;
wire signed [W-1:0] d [1:L]; // Auxilary array
// Chebychev coefficients c1, c2, c3 for 8 bit precision
// c1 = 212; c3 = -12; c5 = 1;

always @(posedge clk or posedge reset) begin
  if (reset) begin // Asynchronous clear
    x <= 0; f_out <= 0;
  end else begin
    x <= x_in;      // FF for input and output
    f_out <= f;
  end
end

// Compute sum-of-products with
// Clenshaw's recurrence formula
assign d[5] = 'sd1; // c5=1
assign d[4] = (x * d[5]) / 128;
assign d[3] = ((x * d[4]) / 128) - d[5] - 12; // c3=-12
assign d[2] = ((x * d[3]) / 128) - d[4];
assign d[1] = ((x * d[2]) / 128) - d[3] + 212; // c1=212
assign f    = ((x * d[1]) / 256) - d[2];
                           // last step is different

assign d_o1 = d[1]; // Provide test signals as outputs
assign d_o2 = d[2];
assign d_o3 = d[3];
assign d_o4 = d[4];
assign d_o5 = d[5];
endmodule

//*****
// IEEE STD 1364-2001 Verilog file: ln.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module ln #(parameter N = 5, // Number of Coefficients-1

```

```

        parameter W= 17) // Bitwidth -1
(input clk,                                // System clock
input reset,                               // Asynchronous reset
input signed [W:0] x_in,                   // System input
output reg signed [W:0] f_out); // System output
// -----
reg signed [W:0] x, f; // Auxilary register
wire signed [W:0] p [0:5];
reg signed [W:0] s [0:5];

// Polynomial coefficients for 16 bit precision:
// f(x) = (1 + 65481 x -32093 x^2 + 18601 x^3
//           -8517 x^4 + 1954 x^5)/65536
assign p[0] = 18'sd1;
assign p[1] = 18'sd65481;
assign p[2] = -18'sd32093;
assign p[3] = 18'sd18601;
assign p[4] = -18'sd8517;
assign p[5] = 18'sd1954;

always @(posedge clk or posedge reset)
begin : Store
    if (reset) begin                  // Asynchronous clear
        x <= 0; f_out <= 0;
    end else begin
        x <= x_in;      // Store input in register
        f_out <= f;
    end
end

always @*          // Compute sum-of-products
begin : SOP
    integer k; // define the loop variable
    reg signed [35:0] slv;

    s[N] = p[N];
// Polynomial Approximation from Chebyshev coefficients
    for (k=N-1; k>=0; k=k-1)
    begin
        slv = x * s[k+1]; // no FFs for slv
        s[k] = (slv >>> 16) + p[k];
    end      // x*s/65536 problem 32 bits
    f <= s[0];      // make visable outside
end

```

```

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: sqrt.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module sqrt           //----> Interface
  (input  clk,          // System clock
   input  reset,         // Asynchronous reset
   output [1:0] count_o, // Counter SLL
   input  signed [16:0] x_in, // System input
   output signed [16:0] pre_o, // Prescaler
   output signed [16:0] x_o,  // Normalized x_in
   output signed [16:0] post_o, // Postscaler
   output signed [3:0] ind_o, // Index to p
   output signed [16:0] imm_o, // ALU preload value
   output signed [16:0] a_o,  // ALU factor
   output signed [16:0] f_o,  // ALU output
   output reg signed [16:0] f_out); // System output
// -----
// Define the operation modes:
parameter load=0, mac=1, scale=2, denorm=3, nop=4;
// Assign the FSM states:
parameter start=0, leftshift=1, sop=2,
          rightshift=3, done=4;

reg [3:0] s, op;
reg [16:0] x; // Auxilary
reg signed [16:0] a, b, f, imm; // ALU data
reg signed [33:0] af; // Product double width
reg [16:0] pre, post;
reg signed [3:0] ind;
reg [1:0] count;
// Chebychev poly coefficients for 16 bit precision:
wire signed [16:0] p [0:4];

assign p[0] = 7563;
assign p[1] = 42299;
assign p[2] = -29129;
assign p[3] = 15813;
assign p[4] = -3778;

always @(posedge reset or posedge clk) //----> SQRT FSM
begin : States                  // sample at clk rate

```

```

if (reset) begin          // Asynchronous reset
    s <= start; f_out <= 0; op <= 0; count <= 0;
    imm <= 0; ind <= 0; a <= 0; x <= 0;
end else begin
    case (s)           // Next State assignments
        start : begin // Initialization step
            s <= leftshift; ind = 4;
            imm <= x_in;           // Load argument in ALU
            op <= load; count = 0;
        end
        leftshift : begin // Normalize to 0.5 .. 1.0
            count = count + 1; a <= pre; op <= scale;
            imm <= p[4];
            if (count == 2) op <= nop;
            if (count == 3) begin // Normalize ready ?
                s <= sop; op <= load; x <= f;
            end
        end
        sop : begin      // Processing step
            ind = ind - 1; a <= x;
            if (ind == -1) begin // SOP ready ?
                s <= rightshift; op <= denorm; a <= post;
            end else begin
                imm <= p[ind]; op <= mac;
            end
        end
        rightshift : begin // Denormalize to original range
            s <= done; op <= nop;
        end
        done : begin      // Output of results
            f_out <= f;           // I/O store in register
            op <= nop;
            s <= start;           // start next cycle
        end
    endcase
end
always @ (posedge reset or posedge clk)
begin : ALU           // Define the ALU operations
    if (reset)           // Asynchronous clear
        f <= 0;
    else begin

```

```

af = a * f;
case (op)
    load   : f  <= imm;
    mac    : f  <= (af >>> 15) + imm;
    scale   : f  <= af;
    denorm : f  <= af >>> 15;
    nop    : f  <= f;
    default : f  <= f;
endcase
end
end

always @(x_in)
begin : EXP
    reg [16:0] slv;
    reg [16:0] po, pr;
    integer K; // Loop variable

    slv = x_in;
    // Compute pre-scaling:
    for (K=0; K <= 15; K= K+1)
        if (slv[K] == 1)
            pre = 1 << (14-K);
    // Compute post scaling:
    po = 1;
    for (K=0; K <= 7; K= K+1) begin
        if (slv[2*K] == 1)      // even 2^k gets 2^k/2
            po = 1 << (K+8);
    // sqrt(2): CSD Error = 0.0000208 = 15.55 effective bits
    // +1 +0. -1 +0 -1 +0 +1 +0 +1 +0 +0 +0 +0 +0 +1
    // 9      7      5      3      1                  -5
        if (slv[2*K+1] == 1) // odd k has sqrt(2) factor
            po = (1<<(K+9)) - (1<<(K+7)) - (1<<(K+5))
                + (1<<(K+3)) + (1<<(K+1)) + (1<<(K-5));
    end
    post <= po;
end

assign a_o = a; // Provide some test signals as outputs
assign imm_o = imm;
assign f_o = f;
assign pre_o = pre;
assign post_o = post;
assign x_o = x;

```

```
assign ind_o = ind;
assign count_o = count;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: magnitude.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module magnitude
  (input clk,           // System clock
   input reset,         // Asynchron reset
   input signed [15:0] x, y,    // System input
   output reg signed [15:0] r); // System output
// -----
  reg signed [15:0] x_r, y_r, ax, ay, mi, ma;
  // Approximate the magnitude via
  // r = alpha*max(|x|,|y|) + beta*min(|x|,|y|)
  // use alpha=1 and beta=1/4

  always @(posedge reset or posedge clk) // Control the
  if (reset) begin          // system sample at clk rate
    x_r <= 0; y_r <= 0; // Asynchronous clear
  end else begin
    x_r <= x; y_r <= y;
  end

  always @* begin
  ax = (x_r>=0)? x_r : -x_r; // Take absolute values first
  ay = (y_r>=0)? y_r : -y_r;

  if (ax > ay) begin // Determine max and min values
    mi = ay;
    ma = ax;
  end else begin
    mi = ax;
    ma = ay;
  end
  end

  always @(posedge reset or posedge clk)
  if (reset)          // Asynchronous clear
    r <= 0;
  else
```

```

r <= ma + mi/4; // Compute r=alpha*max+beta*min

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: fir_gen.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// This is a generic FIR filter generator
// It uses W1 bit data/coefficients bits
module fir_gen
#(parameter W1 = 9,      // Input bit width
         W2 = 18,     // Multiplier bit width 2*W1
         W3 = 19,     // Adder width = W2+log2(L)-1
         W4 = 11,     // Output bit width
         L = 4)      // Filter length
  (input clk,                  // System clock
   input reset,                // Asynchronous reset
   input Load_x,               // Load/run switch
   input signed [W1-1:0] x_in,  // System input
   input signed [W1-1:0] c_in,  //Coefficient data input
   output signed [W4-1:0] y_out); // System output
// -----
  reg signed [W1-1:0] x;
  wire signed [W3-1:0] y;
// 1D array types i.e. memories supported by Quartus
// in Verilog 2001; first bit then vector size
  reg signed [W1-1:0] c [0:3]; // Coefficient array
  wire signed [W2-1:0] p [0:3]; // Product array
  reg signed [W3-1:0] a [0:3]; // Adder array

//----> Load Data or Coefficient
  always @(posedge clk or posedge reset)
    begin: Load
      integer k;      // loop variable
      if (reset) begin          // Asynchronous clear
        for (k=0; k<=L-1; k=k+1) c[k] <= 0;
        x <= 0;
      end else if (! Load_x) begin
        c[3] <= c_in; // Store coefficient in register
        c[2] <= c[3]; // Coefficients shift one
        c[1] <= c[2];
        c[0] <= c[1];
      end else

```

```

        x <= x_in; // Get one data sample at a time
    end

//----> Compute sum-of-products
always @(posedge clk or posedge reset)
begin: SOP
    // Compute the transposed filter additions
    integer k;      // loop variable
    if (reset)      // Asynchronous clear
        for (k=0; k<=3; k=k+1) a[k] <= 0;
    else begin
        a[0] <= p[0] + a[1];
        a[1] <= p[1] + a[2];
        a[2] <= p[2] + a[3];
        a[3] <= p[3]; // First TAP has only a register
    end
end
assign y = a[0];

genvar I; //Define loop variable for generate statement
generate
    for (I=0; I<L; I=I+1) begin : MulGen
        // Instantiate L multipliers
        assign p[I] = x * c[I];
    end
endgenerate

assign y_out = y[W3-1:W3-W4];

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: fir_srg.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module fir_srg                                //----> Interface
    (input clk,                               // System clock
     input reset,                            // Asynchronous reset
     input signed [7:0] x,                  // System input
     output reg signed [7:0] y); // System output
// -----
// Tapped delay line array of bytes
reg signed [7:0] tap [0:3];
integer I; // Loop variable

```

```

always @(posedge clk or posedge reset)
begin : P1 //----> Behavioral Style
  // Compute output y with the filter coefficients weight.
  // The coefficients are [-1 3.75 3.75 -1].
  // Multiplication and division can
  // be done in Verilog 2001 with signed shifts.
  if (reset) begin // Asynchronous clear
    for (I=0; I<=3; I=I+1) tap[I] <= 0;
    y <= 0;
  end else begin
    y <= (tap[1] <<< 1) + tap[1] + (tap[1] >>> 1)- tap[0]
      + ( tap[1] >>> 2) + (tap[2] <<< 1) + tap[2]
      + (tap[2] >>> 1) + (tap[2] >>> 2) - tap[3];

    for (I=3; I>0; I=I-1) begin
      tap[I] <= tap[I-1]; // Tapped delay line: shift one
    end
    tap[0] <= x; // Input in register 0
  end
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: case5p.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module case5p
  (input      clk,
   input [4:0] table_in,
   output reg [4:0] table_out); // range 0 to 25
// -----
  reg [3:0] lsbs;
  reg [1:0] msbs0;
  reg [4:0] table0out00, table0out01;

// These are the distributed arithmetic CASE tables for
// the 5 coefficients: 1, 3, 5, 7, 9

always @(posedge clk) begin
  lsbs[0] = table_in[0];
  lsbs[1] = table_in[1];
  lsbs[2] = table_in[2];

```

```
lsbs[3] = table_in[3];
msbs0[0] = table_in[4];
msbs0[1] = msbs0[0];
end

// This is the final DA MPX stage.
always @(posedge clk) begin
    case (msbs0[1])
        0 : table_out <= table0out00;
        1 : table_out <= table0out01;
        default : ;
    endcase
end

// This is the DA CASE table 00 out of 1.
always @(posedge clk) begin
    case (lsbs)
        0 : table0out00 = 0;
        1 : table0out00 = 1;
        2 : table0out00 = 3;
        3 : table0out00 = 4;
        4 : table0out00 = 5;
        5 : table0out00 = 6;
        6 : table0out00 = 8;
        7 : table0out00 = 9;
        8 : table0out00 = 7;
        9 : table0out00 = 8;
        10 : table0out00 = 10;
        11 : table0out00 = 11;
        12 : table0out00 = 12;
        13 : table0out00 = 13;
        14 : table0out00 = 15;
        15 : table0out00 = 16;
        default ;
    endcase
end

// This is the DA CASE table 01 out of 1.
always @(posedge clk) begin
    case (lsbs)
        0 : table0out01 = 9;
        1 : table0out01 = 10;
        2 : table0out01 = 12;
        3 : table0out01 = 13;
```



```

integer k;

reg [2:0] count;           // Counts the shifts
reg [6:0] p;               // Temporary register

if (reset) begin           // Asynchronous reset
    state <= s0;
    x0 <= 0; x1 <= 0; x2 <= 0; p <= 0; y <= 0;
    count <= 0;
end else
    case (state)
        s0 : begin           // Initialization step
            state <= s1;
            count = 0;
            p <= 0;
            x0 <= x0_in;
            x1 <= x1_in;
            x2 <= x2_in;
        end
        s1 : begin           // Processing step
            if (count == 4) begin// Is sum of product done?
                y <= p;          // Output of result to y and
                state <= s0;       // start next sum of product
            end else begin //Subtract for last accumulator step
                if (count ==3)   // i.e. p/2 +/- table_out * 8
                    p <= (p >>> 1) - (table_out <<< 3);
                else             // Accumulation for all other steps
                    p <= (p >>> 1) + (table_out <<< 3);
                for (k=0; k<=2; k= k+1) begin      // Shift bits
                    x0[k] <= x0[k+1];
                    x1[k] <= x1[k+1];
                    x2[k] <= x2[k+1];
                end
                count = count + 1;
                state <= s1;
            end
        end
    endcase
end

case3s LC_Table0
( .table_in(table_in), .table_out(table_out));

assign lut = table_out; // Provide test signal

```

```
endmodule
```

```

//*****
// IEEE STD 1364-2001 Verilog file: case3s.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module case3s
  (input  [2:0] table_in, // Three bit
   output reg [3:0] table_out); // Range -2 to 4 -> 4 bits
// -----
// This is the DA CASE table for
// the 3 coefficients: -2, 3, 1

  always @(table_in)
  begin
    case (table_in)
      0 :   table_out = 0;
      1 :   table_out = -2;
      2 :   table_out = 3;
      3 :   table_out = 1;
      4 :   table_out = 1;
      5 :   table_out = -1;
      6 :   table_out = 4;
      7 :   table_out = 2;
      default : ;
    endcase
  end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: dapara.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
`include "case3s.v" // User defined component

module dapara           //----> Interface
  (input      clk,        // System clock
   input      reset,       // Asynchron reset
   input signed [3:0] x_in, // System input
   output reg signed[6:0] y); // System output
// -----

```

```

reg signed [2:0] x [0:3];
wire signed [3:0] h [0:3];
reg signed [4:0] s0, s1;
reg signed [3:0] t0, t1, t2, t3;

always @(posedge clk or posedge reset)
begin : DA //----> DA in behavioral style
    integer k,l;
    if (reset) begin // Asynchronous clear
        for (k=0; k<=3; k=k+1) x[k] <= 0;
        y <= 0;
        t0 <= 0; t1 <= 0; t2 <= 0; t3 <= 0; s0 <= 0; s1 <= 0;
    end else begin
        for (l=0; l<=3; l=l+1) begin // For all 4 vectors
            for (k=0; k<=1; k=k+1) begin // shift all bits
                x[1][k] <= x[1][k+1];
            end
        end
        for (k=0; k<=3; k=k+1) begin // Load x_in in the
            x[k][2] <= x_in[k]; // MSBs of the registers
        end
        y <= h[0] + (h[1] <<< 1) + (h[2] <<< 2) - (h[3] <<< 3);
    // Sign extensions, pipeline register, and adder tree:
    //      t0 <= h[0]; t1 <= h[1]; t2 <= h[2]; t3 <= h[3];
    //      s0 <= t0 + (t1 <<< 1);
    //      s1 <= t2 - (t3 <<< 1);
    //      y <= s0 + (s1 <<< 2);
    end
end

genvar i;// Need to declare loop variable in Verilog 2001
generate // One table for each bit in x_in
    for (i=0; i<=3; i=i+1) begin:LC_Tables
        case3s LC_Table0 (.table_in(x[i]), .table_out(h[i]));
    end
endgenerate

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: iir.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module iir #(parameter W = 14) // Bit width - 1

```

```

    (input clk,           // System clock
     input reset,        // Asynchronous reset
     input signed [W:0] x_in, // System input
     output signed [W:0] y_out); // System output
// -----
// reg signed [W:0] x, y;

// Use FFs for input and recursive part
always @ (posedge clk or posedge reset)
  if (reset) begin           // Note: there is a signed
    x <= 0; y <= 0;          // integer in Verilog 2001
  end else begin
    x  <= x_in;
    y  <= x + (y >>> 1) + (y >>> 2); // >>> uses less LEs
    // y  <= x + y / 'sd2 + y / 'sd4; // same as /2 and /4
    //y  <= x + y / 2 + y / 4; // div with / uses more LEs
    //y <= x + {y[W],y[W:1]}+ {y[W],y[W],y[W:2]};
  end

assign y_out = y;           // Connect y to output pins

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: iir_pipe.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module iir_pipe           //----> Interface
  #(parameter W = 14)      // Bit width - 1
  (input clk,               // System clock
   input reset,             // Asynchronous reset
   input signed [W:0] x_in, // System input
   output signed [W:0] y_out); // System output
// -----
reg signed [W:0] x, x3, sx;
reg signed [W:0] y, y9;

always @ (posedge clk or posedge reset) // Infer FFs for
begin           // input, output and pipeline stages;
  if (reset) begin // Asynchronous clear
    x <= 0; x3 <= 0; sx <= 0; y9 <= 0; y <= 0;
  end else begin
    x  <= x_in;      // use non-blocking FF assignments
  end
end

```

```

    x3  <= (x >>> 1) + (x >>> 2);           // i.e. x / 2 + x / 4 = x*3/4
    sx  <= x + x3;//Sum of x element i.e. output FIR part
    y9  <= (y >>> 1) + (y >>> 4);           // i.e. y / 2 + y / 16 = y*9/16
    y   <= sx + y9;                            // Compute output
  end
end

assign y_out = y ; // Connect register y to output pins

endmodule

```

```

//*****
// IEEE STD 1364-2001 Verilog file: iir_par.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module iir_par      //---> Interface
#(parameter W = 14) // bit width - 1
  (input  clk,          // System clock
   input  reset,         // Asynchronous reset
   input signed [W:0] x_in,    // System input
   output signed [W:0] x_e, x_o,// Even/odd input x_in
   output signed [W:0] y_e, y_o,// Even/odd output y_out
   output clk2,          // Clock divided by 2
   output signed [W:0] y_out); // System output
// -----
  reg signed [W:0] x_even, xd_even, x_odd, xd_odd, x_wait;
  reg signed [W:0] y_even, y_odd, y_wait, y;
  reg signed [W:0] sum_x_even, sum_x_odd;
  reg     clk_div2;
  reg [0:0] state;

  always @ (posedge clk or posedge reset) // Clock divider
  begin : clk_divider                // by 2 for input clk
    if (reset) clk_div2 <= 0;
    else      clk_div2 <= ! clk_div2;
  end

  always @ (posedge clk or posedge reset) // Split x into
  begin : Multiplex                  // even and odd samples;
    parameter even=0, odd=1;           // recombine y at clk rate

```

```

if (reset) begin                                // Asynchronous reset
    state <= even; x_even <= 0; x_odd <= 0;
    y <= 0; x_wait <= 0; y_wait <= 0;
end else
case (state)
even : begin
    x_even <= x_in;
    x_odd <= x_wait;
    y <= y_wait;
    state <= odd;
end
odd : begin
    x_wait <= x_in;
    y <= y_odd;
    y_wait <= y_even;
    state <= even;
end
endcase
end

assign y_out = y;
assign clk2 = clk_div2;
assign x_e = x_even; // Monitor some extra test signals
assign x_o = x_odd;
assign y_e = y_even;
assign y_o = y_odd;

always @ (negedge clk_div2 or posedge reset)
begin: Arithmetic
if (reset) begin
    xd_even <= 0; sum_x_even <= 0; y_even <= 0;
    xd_odd <= 0; sum_x_odd <= 0; y_odd <= 0;
end else begin
    xd_even <= x_even;
    sum_x_even <= x_odd + (xd_even >>> 1)+(xd_even >>> 2);
        // i.e. x_odd + x_even/2 + x_even /4
    y_even <= sum_x_even + (y_even >>> 1)+(y_even >>> 4);
        // i.e. sum_x_even + y_even / 2 + y_even /16
    xd_odd <= x_odd;
    sum_x_odd <= xd_even + (xd_odd >>> 1)+(xd_odd >>> 4);
        // i.e. x_even + xd_odd / 2 + xd_odd /4
    y_odd <= sum_x_odd + (y_odd >>> 1)+(y_odd >>> 4);
        // i.e. sum_x_odd + y_odd / 2 + y_odd / 16
end

```

```

    end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: iir5sfix.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// 5th order IIR in direct form implementation
module iir5sfix
  (input clk,           // System clock
   input reset,         // Asynchron reset
   input switch,        // Feedback switch
   input signed [63:0] x_in, // System input
   output signed [39:0] t_out, // Feedback
   output signed [39:0] y_out); // System output
// -----
  wire signed[63:0] a2, a3, a4, a5, a6; // Feedback A
  wire signed[63:0] b1, b2, b3, b4, b5, b6; // Feedforward B
  wire signed [63:0] h;
  reg signed [63:0] s1, s2, s3, s4;
  reg signed [63:0] y, t, r2, r3, r4;
  reg signed [127:0] a6h, a5h, a4h, a3h, a2h;
  reg signed [127:0] b6h, b5h, b4h, b3h, b2h, b1h;

  // Feedback A scaled by 2^30
  assign a2 = 64'h000000013DF707FA; // (-)4.9682025852
  assign a3 = 64'h0000000277FBF6D7; // 9.8747536754
  assign a4 = 64'h00000002742912B6; // (-)9.8150069021
  assign a5 = 64'h00000001383A6441; // 4.8785639415
  assign a6 = 64'h000000003E164061; // (-)0.9701081227
  // Feedforward B scaled by 2^30
  assign b1 = 64'h000000000004F948; // 0.0003035737
  assign b2 = 64'h00000000000EE2A2; // (-)0.0009085259
  assign b3 = 64'h000000000009E95E; // 0.0006049556
  assign b4 = 64'h000000000009E95E; // 0.0006049556
  assign b5 = 64'h00000000000EE2A2; // (-)0.0009085259
  assign b6 = 64'h000000000004F948; // 0.0003035737
  assign h = (switch) ? x_in-t // Switch is closed
                     : x_in; // Switch is open

  always @ (posedge clk or posedge reset)
  begin : P1 // First equations without infering registers
    if (reset) begin // Asynchronous clear

```

```

t <= 0; y <= 0; r2 <= 0; r3 <= 0; r4 <= 0; s1 <= 0;
s2 <= 0; s3 <= 0; s4 <= 0; a6h <= 0; b6h <= 0;
end else begin // IIR filter in direct form
    a6h <= a6 * h; // h*a[6] use register
    a5h = a5 * h; // h*a[5]
    r4 <= (a5h >>> 30) - (a6h >>> 30); // h*a[5]+r5
    a4h = a4 * h; // h*a[4]
    r3 <= r4 - (a4h >>> 30); // h*a[4]+r4
    a3h = a3 * h; // h*a[3]
    r2 <= r3 + (a3h >>> 30); // h*a[3]+r3
    a2h = a2 * h; // h*a[2]+r2
    t <= r2 - (a2h >>> 30); // h*a[2]+r2
    b6h <= b6 * h; // h*b[6] use register
    b5h = b5 * h; // h*b[5]+s5 no register
    s4 <= (b6h >>> 30) - (b5h >>> 30); // h*b[5]+s5
    b4h = b4 * h; // h*b[4]+s4
    s3 <= s4 + (b4h >>> 30); // h*b[4]+s4
    b3h = b3 * h; // h*b[3]
    s2 <= s3 + (b3h >>> 30); // h*b[3]+s3
    b2h = b2 * h; // h*b[2]
    s1 <= s2 - (b2h >>> 30); // h*b[2]+s2
    b1h = b1 * h; // h*b[1]
    y <= s1 + (b1h >>> 30); // h*b[1]+s1
end
end

// Redefine bits as 40 bit SLV
// Change 30 to 16 bit fraction, i.e. cut 14 LSBs
assign y_out = y[53:14];
assign t_out = t[53:14];

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: iir5para.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// Description: 5th order IIR parallel form implementation
// Coefficients:
// D = 0.00030357
// B1 = 0.0031 -0.0032 0
// A1 = 1.0000 -1.9948 0.9959
// B2 = -0.0146 0.0146 0
// A2 = 1.0000 -1.9847 0.9852
// B3 = 0.0122

```

```

// A3 = 0.9887
// -----
module iir5para           // I/O data in 16.16 format
  (input clk,               // System clock
   input reset,             // Asynchronous reset
   input signed [31:0] x_in, // System input
   output signed [31:0] y_Dout, // 0 order
   output signed [31:0] y_1out, // 1. order
   output signed [31:0] y_21out, // 2. order 1
   output signed [31:0] y_22out, // 2. order 2
   output signed [31:0] y_out); // System output
// -----
// Internal type has 2 bits integer and 18 bits fraction:
reg signed [19:0] s11, r13, r12; // 1. BiQuad regs.
reg signed [19:0] s21, r23, r22; // 2. BiQuad regs.
reg signed [19:0] r32, r41, r42, r43, y;
reg signed [19:0] x;

// Products have double bit width
reg signed [39:0] b12x, b11x, a13r12, a12r12; // 1. BiQuad
reg signed [39:0] b22x, b21x, a23r22, a22r22; // 2. BiQuad
reg signed [39:0] b31x, a32r32, Dx;

// All coefficients use 5*4=20 bits and are scaled by 2^18
wire signed [19:0] a12, a13, b11, b12; // First BiQuad
wire signed [19:0] a22, a23, b21, b22; // Second BiQuad
wire signed [19:0] a32, b31, D; // First order and direct
// First BiQuad coefficients
  assign a12 = 20'h7FAB9; // (-)1.99484680
  assign a13 = 20'h3FBDO; // 0.99591112
  assign b11 = 20'h00340; // 0.00307256
  assign b12 = 20'h00356; // (-)0.00316061
// Second BiQuad coefficients
  assign a22 = 20'h7F04F; // (-)1.98467605
  assign a23 = 20'h3F0E4; // 0.98524428
  assign b21 = 20'h00F39; // (-)0.01464265
  assign b22 = 20'h00F38; // 0.01464684
// First order system with R(5) and P(5)
  assign a32 = 20'h3F468; // 0.98867974
  assign b31 = 20'h00C76; // 0.012170
// Direct system
  assign D = 20'h0004F; // 0.000304

always @ (posedge clk or posedge reset)

```

```

begin : P1
  if (reset) begin           // Asynchronous clear
    b12x <= 0; s11 <= 0; r13 <= 0; r12 <= 0;
    b22x <= 0; s21 <= 0; r23 <= 0; r22 <= 0;
    b31x <= 0; r32 <= 0; r41 <= 0;
    r42 <= 0; r43 <= 0; y <= 0; x <= 0;
  end else begin           // SOS modified BiQuad form
    // Redefine bits as FIX 16.16 number to
    x <= {x_in[17:0], 2'b00};
    // Internal precision 2.19 format, i.e. 21 bits
    // 1. BiQuad is 2. order
    b12x <= b12 * x;
    b11x = b11 * x;
    s11 <= (b11x >>> 18) - (b12x >>> 18); // was +
    a13r12 = a13 * r12;
    r13 <= s11 - (a13r12 >>> 18);
    a12r12 = a12 * r12;
    r12 <= r13 + (a12r12 >>> 18); // was -
    // 2. BiQuad is 2. order
    b22x <= b22 * x;
    b21x = b21 * x;
    s21 <= (b22x >>> 18) - (b21x >>> 18); // was +
    a23r22 = a23 * r22;
    r23 <= s21 - (a23r22 >>> 18);
    a22r22 = a22 * r22;
    r22 <= r23 + (a22r22 >>> 18); // was -
    // 3. Section is 1. order
    b31x <= b31 * x;
    a32r32 = a32 * r32;
    r32 <= (b31x >>> 18) + (a32r32 >>> 18);
    // 4. Section is assign
    Dx = D * x;
    r41 <= Dx >>> 18;
    // Output adder tree
    r42 <= r41;
    r43 <= r42 + r32;
    y <= r12 + r22 + r43;
  end
end

// Change 19 to 16 bit fraction, i.e. cut 2 LSBs
// Redefine bits as 32 bit SLV
assign y_out    = {{14{y[19]}},y[19:2]};
assign y_Dout   = {{14{r42[19]}},r42[19:2]};

```

```

assign y_1out = {{14{r32[19]}},r32[19:2]};
assign y_21out = {{14{r22[19]}},r22[19:2]};
assign y_22out = {{14{r12[19]}},r12[19:2]};

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: iir5lwdf.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// Description: 5th order Lattice Wave Digital Filter
// Coefficients gamma:
// 0.988727 -0.000528 -1.995400 -0.000282 -1.985024
// -----
module iir5lwdf                                //----> Interface
  (input clk,                                     // System clock
   input reset,                                    // Asynchronous reset
   input signed [31:0] x_in,                      // System input
   output signed [31:0] y_ap1out, // AP1 out
   output signed [31:0] y_ap2out, // AP2 out
   output signed [31:0] y_ap3out, // AP3 out
   output signed [31:0] y_out); // System output
// -----
// Internal signals have 7.15 format
reg signed [21:0] c1, c2, c3, l2, l3;
reg signed [21:0] a4, a5, a6, a8, a9, a10;
reg signed [21:0] x, ap1, ap2, ap3, ap3r, y;

// Products have double bit width
reg signed [41:0] p1, a4g2, a4g3, a8g4, a8g5;

//Coefficients gamma use 5*4=20 bits and are scaled by 2^15
wire signed [19:0] g1, g2, g3, g4, g5;
assign g1 = 20'h07E8F; // 0.988739
assign g2 = 20'h00011; // (-)0.000519
assign g3 = 20'h0FF69; // (-)1.995392
assign g4 = 20'h00009; // (-)0.000275
assign g5 = 20'h0FE15; // (-)1.985016

always @ (posedge clk or posedge reset)
begin : P1
  if (reset) begin // Asynchronous clear
    c1 <= 0; ap1 <= 0; c2 <= 0; l2 <= 0;
    ap2 <= 0; c3 <= 0; l3 <= 0; ap3 <= 0;
    ap3r <= 0; y <= 0; x <= 0;
  end
  else begin
    l2 = x;
    ap1 = a4 * g1 + a5 * g2 + a6 * g3 + a8 * g4 + a9 * g5;
    ap2 = ap1 * g2;
    ap3 = ap2 * g3;
    ap3r = ap3 * g4;
    y = ap3r + a10 * g5;
  end
end

```

```

    end else begin // AP LWDF form
      // Redefine 16.16 input bits as internal precision
      // in 7.15 format, i.e. 20 bits
      x <= x_in[22:1];
    // 1. AP section is 1. order
      p1 = g1 * (c1 - x);
      c1 <= x + (p1 >>> 15);
      ap1 <= c1 + (p1 >>> 15);
    // 2. AP section is 2. order
      a4 = ap1 - 12 + c2 ;
      a4g2 = a4 * g2;
      a5 = c2 - (a4g2 >>> 15); // was +
      a4g3 = a4 * g3;
      a6 = -(a4g3 >>> 15) - 12; // was +
      c2 <= a5;
      12 <= a6;
      ap2 <= -a5 - a6 - a4;
    // 3. AP section is 2. order
      a8 = x - 13 + c3;
      a8g4 = a8 * g4;
      a9 = c3 - (a8g4 >>> 15); // was +
      a8g5 = a8 * g5;
      a10 = -(a8g5 >>> 15) - 13; // was +
      c3 <= a9;
      13 <= a10;
      ap3 <= -a9 - a10 - a8;
      ap3r <= ap3; // Extra register due to AP1
    // Output adder
      y <= ap3r + ap2;
    end
  end

  // Change 15 to 16 bit fraction, i.e. add 1 LSBs
  // Redefine bits as 32 bit SLV 1+22+9=32
  assign y_out    = {{9{y[21]}},y,1'b0};
  assign y_ap1out = {{9{ap1[21]}},ap1,1'b0};
  assign y_ap2out = {{9{ap2[21]}},ap2,1'b0};
  assign y_ap3out = {{9{ap3r[21]}},ap3r,1'b0};

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: db4poly.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org

```

```

//*****
module db4poly          //----> Interface
  (input clk,           // System clock
   input reset,         // Asynchron reset
   input signed [7:0] x_in,    // System input
   output clk2,          // Clock divided by 2
   output signed [16:0] x_e, x_o, // Even/odd x_in
   output signed [16:0] g0, g1,  // Poly filter 0/1
   output signed [8:0] y_out); // System output
// -----
  reg signed [7:0] x_odd, x_even, x_wait;
  reg clk_div2;

// Register for multiplier, coefficients, and taps
  reg signed [16:0] m0, m1, m2, m3, r0, r1, r2, r3;
  reg signed [16:0] x33, x99, x107;
  reg signed [16:0] y;

  always @ (posedge clk or posedge reset) // Split into even
  begin : Multiplex           // and odd samples at clk rate
    parameter even=0, odd=1;
    reg [0:0] state;

    if (reset) begin           // Asynchronous reset
      state <= even;
      clk_div2 = 0; x_even <= 0; x_odd <= 0; x_wait <= 0;
    end else
      case (state)
        even : begin
          x_even <= x_in;
          x_odd  <= x_wait;
          clk_div2 = 1;
          state <= odd;
        end
        odd : begin
          x_wait <= x_in;
          clk_div2 = 0;
          state <= even;
        end
      endcase
    end

    always @ (x_odd, x_even)
    begin : RAG

```

```

// Compute auxiliary multiplications of the filter
  x33 = (x_odd <<< 5) + x_odd;
  x99 = (x33 <<< 1) + x33;
  x107 = x99 + (x_odd << 3);
// Compute all coefficients for the transposed filter
  m0 = (x_even <<< 7) - (x_even <<< 2); // m0 = 124
  m1 = x107 <<< 1; // m1 = 214
  m2 = (x_even <<< 6) - (x_even <<< 3)
    + x_even; // m2 = 57
  m3 = x33; // m3 = -33
end

always @ (posedge reset or negedge clk_div2)
begin : AddPolyphase // use non-blocking assignments
  if (reset) begin // Asynchronous clear
    r0 <= 0; r1 <= 0; r2 <= 0; r3 <= 0;
    y <= 0;
  end else begin
    //----- Compute filter G0
    r0 <= r2 + m0; // g0 = 128
    r2 <= m2; // g2 = 57
    //----- Compute filter G1
    r1 <= -r3 + m1; // g1 = 214
    r3 <= m3; // g3 = -33
    // Add the polyphase components
    y <= r0 + r1;
  end
end

// Provide some test signals as outputs
assign x_e = x_even;
assign x_o = x_odd;
assign clk2 = clk_div2;
assign g0 = r0;
assign g1 = r1;

assign y_out = y >>> 8; // Connect y / 256 to output

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: cic3r32.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module cic3r32 //----> Interface

```

```

(input      clk,          // System clock
 input      reset,        // Asynchronous reset
 input signed [7:0] x_in, // System input
 output signed [9:0] y_out, // System output
 output reg clk2);       // Clock divider
// -----
parameter hold=0, sample=1;
reg [1:0] state;
reg [4:0] count;
reg signed [7:0] x;      // Registered input
reg signed [25:0] i0, i1 , i2; // I section 0, 1, and 2
reg signed [25:0] i2d1, i2d2, c1, c0;           // I + COMB 0
reg signed [25:0] c1d1, c1d2, c2;             // COMB section 1
reg signed [25:0] c2d1, c2d2, c3;             // COMB section 2

always @(posedge clk or posedge reset)
begin : FSM
    if (reset) begin           // Asynchronous reset
        count <= 0;
        state <= hold;
        clk2 <= 0;
    end else begin
        if (count == 31) begin
            count <= 0;
            state <= sample;
            clk2 <= 1;
        end else begin
            count <= count + 1;
            state <= hold;
            clk2 <= 0;
        end
    end
end

always @(posedge clk or posedge reset)
begin : Int      // 3 stage integrator sections
    if (reset) begin // Asynchronous clear
        x <= 0; i0 <= 0; i1 <= 0; i2 <= 0;
    end else begin
        x     <= x_in;
        i0    <= i0 + x;
        i1    <= i1 + i0 ;
        i2    <= i2 + i1 ;
    end
end

```

```

    end

    always @(posedge clk or posedge reset)
    begin : Comb // 3 stage comb sections
        if (reset) begin // Asynchronous clear
            c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0;
            i2d1 <= 0; i2d2 <= 0; c1d1 <= 0; c1d2 <= 0;
            c2d1 <= 0; c2d2 <= 0;
        end else if (state == sample) begin
            c0    <= i2;
            i2d1 <= c0;
            i2d2 <= i2d1;
            c1    <= c0 - i2d2;
            c1d1 <= c1;
            c1d2 <= c1d1;
            c2    <= c1 - c1d2;
            c2d1 <= c2;
            c2d2 <= c2d1;
            c3    <= c2 - c2d2;
        end
    end

    assign y_out = c3[25:16];

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: cic3s32.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module cic3s32 //----> Interface
    (input      clk,           // System clock
     input      reset,         // Asynchronous reset
     input signed [7:0] x_in,   // System input
     output signed [9:0] y_out, // System output
     output reg  clk2);       // Clock divider
// -----
parameter hold=0, sample=1;
reg [1:0] state;
reg [4:0] count;
reg signed [7:0] x;           // Registered input
reg signed [25:0] i0;          // I section 0
reg signed [20:0] i1;          // I section 1
reg signed [15:0] i2;          // I section 2
reg signed [13:0] i2d1, i2d2, c1, c0; // I+C0

```

```

reg signed [12:0] c1d1, c1d2, c2;           // COMB 1
reg signed [11:0] c2d1, c2d2, c3;           // COMB 2

always @(posedge clk or posedge reset)
begin : FSM
    if (reset) begin          // Asynchronous reset
        count <= 0;
        state <= hold;
        clk2 <= 0;
    end else begin
        if (count == 31) begin
            count <= 0;
            state <= sample;
            clk2 <= 1;
        end
        else begin
            count <= count + 1;
            state <= hold;
            clk2 <= 0;
        end
    end
end

always @(posedge clk or posedge reset)
begin : Int      // 3 stage integrator sections
if (reset) begin // Asynchronous clear
    x <= 0; i0 <= 0; i1 <= 0; i2 <= 0;
end else begin
    x <= x_in;
    i0 <= i0 + x;
    i1 <= i1 + i0[25:5];
    i2 <= i2 + i1[20:5];
end
end

always @(posedge clk or posedge reset)

begin : Comb          // 3 stage comb sections
if (reset) begin // Asynchronous clear
    c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0;
    i2d1 <= 0; i2d2 <= 0; c1d1 <= 0; c1d2 <= 0;
    c2d1 <= 0; c2d2 <= 0;
end else if (state == sample) begin
    c0 <= i2[15:2];

```

```

    i2d1 <= c0;
    i2d2 <= i2d1;
    c1    <= c0 - i2d2;
    c1d1 <= c1[13:1];
    c1d2 <= c1d1;
    c2    <= c1[13:1] - c1d2;
    c2d1 <= c2[12:1];
    c2d2 <= c2d1;
    c3    <= c2[12:1] - c2d2;
end
end

assign y_out = c3[11:2];

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: rc_sinc.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module rc_sinc #(parameter OL = 2, //Output buffer length-1
                  IL = 3,   //Input buffer length -1
                  L   = 10) // Filter length -1
  (input clk,           // System clock
   input reset,         // Asynchronous reset
   input signed [7:0] x_in, // System input
   output [3:0] count_o, // Counter FSM
   output ena_in_o,     // Sample input enable
   output ena_out_o,    // Shift output enable
   output ena_io_o,     // Enable transfer2output
   output signed [8:0] f0_o, // First Sinc filter output
   output signed [8:0] f1_o, // Second Sinc filter output
   output signed [8:0] f2_o, // Third Sinc filter output
   output signed [8:0] y_out); // System output
// -----
reg [3:0] count; // Cycle R_1*R_2
reg ena_in, ena_out, ena_io; // FSM enables
reg signed [7:0] x [0:10]; // TAP registers for 3 filters
reg signed [7:0] ibuf [0:3]; // TAP in registers
reg signed [7:0] obuf [0:2]; // TAP out registers
reg signed [8:0] f0, f1, f2; // Filter outputs

// Constant arrays for multiplier and taps:
wire signed [8:0] c0 [0:10];
wire signed [8:0] c2 [0:10];

```

```

// filter coefficients for filter c0
assign c0[0] = -19; assign c0[1] = 26; assign c0[2]=-42;
assign c0[3] = 106; assign c0[4] = 212; assign c0[5]=-53;
assign c0[6] = 29; assign c0[7] = -21; assign c0[8]=16;
assign c0[9] = -13; assign c0[10] = 11;

// filter coefficients for filter c2
assign c2[0] = 11; assign c2[1] = -13;assign c2[2] = 16;
assign c2[3] = -21;assign c2[4] = 29; assign c2[5] = -53;
assign c2[6] = 212;assign c2[7] = 106;assign c2[8] = -42;
assign c2[9] = 26; assign c2[10] = -19;

always @(posedge reset or posedge clk)
begin : FSM // Control the system and sample at clk rate
  if (reset)           // Asynchronous reset
    count <= 0;
  else
    if (count == 11) count <= 0;
    else               count <= count + 1;
end

always @(posedge clk)
begin      // set the enable signal for the TAP lines
  case (count)
    2, 5, 8, 11 : ena_in <= 1;
    default       : ena_in <= 0;
  endcase

  case (count)
    4, 8      : ena_out <= 1;
    default   : ena_out <= 0;
  endcase

  if (count == 0) ena_io <= 1;
  else           ena_io <= 0;
end

always @(posedge clk or posedge reset)// Input delay line
begin : INPUTMUX
  integer I;      // loop variable
  if (reset)       // Asynchronous clear
    for (I=0; I<=IL; I=I+1) ibuf[I] <= 0;
  else if (ena_in) begin

```

```

        for (I=IL; I>=1; I=I-1)
            ibuf[I] <= ibuf[I-1];           // shift one
            ibuf[0] <= x_in;              // Input in register 0
        end
    end

always @(posedge clk or posedge reset)//Output delay line
begin : OUPUTMUX
    integer I;      // loop variable
    if (reset)       // Asynchronous clear
        for (I=0; I<=0L; I=I+1) obuf[I] <= 0;
    else begin
        if (ena_io) begin // store 3 samples in output buffer
            obuf[0] <= f0;
            obuf[1] <= f1;
            obuf[2] <= f2;
        end else if (ena_out) begin
            for (I=0L; I>=1; I=I-1)
                obuf[I] <= obuf[I-1];      // shift one
        end
    end
end

always @(posedge clk or posedge reset)
begin : TAP          // get 4 samples at one time
    integer I;      // loop variable
    if (reset)       // Asynchronous clear
        for (I=0; I<=10; I=I+1) x[I] <= 0;
    else if (ena_io) begin      // One tapped delay line
        for (I=0; I<=3; I=I+1)
            x[I] <= ibuf[I];    // take over input buffer

        for (I=4; I<=10; I=I+1) // 0->4; 4->8 etc.
            x[I] <= x[I-4];      // shift 4 taps
    end
end

always @(posedge clk or posedge reset)
begin : SOP0         // Compute sum-of-products for f0
    reg signed [16:0] sum; // temp sum
    reg signed [16:0] p [0:10]; // temp products
    integer I;

    for (I=0; I<=L; I=I+1) // Infer L+1 multiplier

```

```

p[I] = c0[I] * x[I];

sum = p[0];
for (I=1; I<=L; I=I+1)      // Compute the direct
    sum = sum + p[I];        // filter adds
if (reset) f0 <= 0;          // Asynchronous clear
else f0 <= sum >>> 8;
end

always @(posedge clk or posedge reset)
begin : SOP1                  // Compute sum-of-products for f1
    if (reset) f1 <= 0;        // Asynchronous clear
    else f1 <= x[5];         // No scaling, i.e., unit impulse
end

always @(posedge clk) // Compute sum-of-products for f2
begin : SOP2
    reg signed[16:0] sum; // temp sum
    reg signed [16:0] p [0:10]; // temp products
    integer I;

    for (I=0; I<=L; I=I+1) // Infer L+1 multiplier
        p[I] = c2[I] * x[I];

    sum = p[0];
    for (I=1; I<=L; I=I+1)      // Compute the direct
        sum = sum + p[I];        // filter adds

    if (reset) f2 <= 0;        // Asynchronous clear
    else f2 <= sum >>> 8;
end

// Provide some test signals as outputs
assign f0_o = f0;
assign f1_o = f1;
assign f2_o = f2;
assign count_o = count;
assign ena_in_o = ena_in;
assign ena_out_o = ena_out;
assign ena_io_o = ena_io;

assign y_out = obuf[OL]; // Connect to output

endmodule

```

```
//*****
// IEEE STD 1364-2001 Verilog file: farrow.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module farrow #(parameter IL = 3) // Input buffer length -1
    (input clk,                                // System clock
     input reset,                               // Asynchronous reset
     input signed [7:0] x_in,                   // System input
     output [3:0] count_o,                      // Counter FSM
     output ena_in_o,                          // Sample input enable
     output ena_out_o,                         // Shift output enable
     output signed [8:0] c0_o,c1_o,c2_o,c3_o, // Phase delays
     output [8:0] d_out,                        // Delay used
     output reg signed [8:0] y_out);           // System output
// -----
reg [3:0] count; // Cycle R_1*R_2
wire [6:0] delta; // Increment d
reg ena_in, ena_out; // FSM enables
reg signed [7:0] x [0:3];
reg signed [7:0] ibuf [0:3]; // TAP registers
reg [8:0] d; // Fractional Delay scaled to 8 bits
// Lagrange matrix outputs:
reg signed [8:0] c0, c1, c2, c3;

assign delta = 85;

always @(posedge reset or posedge clk)      // Control the
begin : FSM                                // system and sample at clk rate
    reg [8:0] dnew;
    if (reset) begin                         // Asynchronous reset
        count <= 0;
        d <= delta;
    end else begin
        if (count == 11)
            count <= 0;
        else
            count <= count + 1;
        if (ena_out) begin                  // Compute phase delay
            dnew = d + delta;
            if (dnew >= 255)
                d <= 0;
            else
                d <= dnew;
        end
    end
end
```

```

        end
    end
end

always @(posedge clk or posedge reset)
begin           // Set the enable signals for the TAP lines
    case (count)
        2, 5, 8, 11 : ena_in <= 1;
        default      : ena_in <= 0;
    endcase

    case (count)
        3, 7, 11 : ena_out <= 1;
        default   : ena_out <= 0;
    endcase
end

always @(posedge clk or posedge reset)
begin : TAP           //----> One tapped delay line
    integer I;      // loop variable
    if (reset)       // Asynchronous clear
        for (I=0; I<=IL; I=I+1) ibuf[I] <= 0;
    else if (ena_in) begin
        for (I=1; I<=IL; I=I+1)
            ibuf[I-1] <= ibuf[I]; // Shift one

        ibuf[IL] <= x_in;          // Input in register IL
    end
end

always @(posedge clk or posedge reset)
begin : GET           // Get 4 samples at one time
    integer I;      // loop variable
    if (reset)       // Asynchronous clear
        for (I=0; I<=IL; I=I+1) x[I] <= 0;
    else if (ena_out) begin
        for (I=0; I<=IL; I=I+1)
            x[I] <= ibuf[I]; // take over input buffer
    end
end

// Compute sum-of-products:
always @(posedge clk or posedge reset)
begin : SOP

```

```

reg signed [8:0] y1, y2, y3; // temp's

// Matrix multiplier iV=inv(Vandermonde) c=iV*x(n-1:n+2)'
//      x(0)    x(1)        x(2)        x(3)
// iV=    0     1.0000       0         0
//   -0.3333   -0.5000    1.0000   -0.1667
//   0.5000   -1.0000    0.5000       0
//  -0.1667    0.5000   -0.5000    0.1667
if (reset) begin           // Asynchronous clear
  y_out <= 0;
  c0 <= 0; c1 <= 0; c2<= 0; c3 <= 0;
end else if (ena_out) begin
  c0 <= x[1];
  c1 <= (-85 * x[0] >>> 8) - (x[1]/2) + x[2] -
                                (43 * x[3] >>> 8);
  c2 <= ((x[0] + x[2]) >>> 1) - x[1];
  c3 <= ((x[1] - x[2]) >>> 1) +
        (43 * (x[3] - x[0]) >>> 8);

// Farrow structure = Lagrange with Horner schema
// for u=0:3, y=y+f(u)*d^u; end;
  y1 = c2 + ((c3 * d) >>> 8); // d is scale by 256
  y2 = ((y1 * d) >>> 8) + c1;
  y3 = ((y2 * d) >>> 8) + c0;

  y_out <= y3; // Connect to output + store in register
end
end

assign c0_o = c0; // Provide test signals as outputs
assign c1_o = c1;
assign c2_o = c2;
assign c3_o = c3;
assign count_o = count;
assign ena_in_o = ena_in;
assign ena_out_o = ena_out;
assign d_out = d;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: cmoms.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****

```

```

module cmoms #(parameter IL = 3) // Input buffer length -1
  (input clk, // System clock
   input reset, // Asynchron reset
   output [3:0] count_o, // Counter FSM
   output ena_in_o, // Sample input enable
   output ena_out_o, // Shift output enable
   input signed [7:0] x_in, // System input
   output signed [8:0] xiir_o, // IIR filter output
   output signed [8:0] c0_o,c1_o,c2_o,c3_o,// C-MOMS matrix
   output signed [8:0] y_out); // System output
// -----
reg [3:0] count; // Cycle R_1*R_2
reg [1:0] t;
reg ena_in, ena_out; // FSM enables
reg signed [7:0] x [0:3];
reg signed [7:0] ibuf [0:3]; // TAP registers
reg signed [8:0] xiir; // iir filter output

reg signed [16:0] y, y0, y1, y2, y3, h0, h1; // temp's

// Spline matrix output:
reg signed [8:0] c0, c1, c2, c3;

// Precomputed value for d**k :
wire signed [8:0] d1 [0:2];
wire signed [8:0] d2 [0:2];
wire signed [8:0] d3 [0:2];

assign d1[0] = 0; assign d1[1] = 85; assign d1[2] = 171;
assign d2[0] = 0; assign d2[1] = 28; assign d2[2] = 114;
assign d3[0] = 0; assign d3[1] = 9; assign d3[2] = 76;

always @ (posedge reset or posedge clk) // Control the
begin : FSM // system sample at clk rate
  if (reset) begin // Asynchronous reset
    count <= 0;
    t <= 1;
  end else begin
    if (count == 11)
      count <= 0;
    else
      count <= count + 1;
    if (ena_out)

```

```

        if (t>=2)      // Compute phase delay
            t <= 0;
        else
            t <= t + 1;
    end
end
assign t_out = t;

always @(posedge clk) // set the enable signal
begin                  // for the TAP lines
    case (count)
        2, 5, 8, 11 : ena_in <= 1;
        default       : ena_in <= 0;
    endcase

    case (count)
        3, 7, 11     : ena_out <= 1;
        default       : ena_out <= 0;
    endcase
end

// Coeffs: H(z)=1.5/(1+0.5z^-1)
always @(posedge clk or posedge reset)
begin : IIR // Compute iir coefficients first
    reg signed [8:0] x1;      // x * 1
    if (reset) begin // Asynchronous clear
        xiir <= 0; x1 <= 0;
    end else
        if (ena_in) begin
            xiir <= (3 * x1 >>> 1) - (xiir >>> 1);
            x1 = x_in;
        end
    end

always @(posedge clk or posedge reset)
begin : TAP           //----> One tapped delay line
    integer I;      // Loop variable
    if (reset) begin // Asynchronous clear
        for (I=0; I<=IL; I=I+1) ibuf[I] <= 0;
    end else
        if (ena_in) begin
            for (I=1; I<=IL; I=I+1)
                ibuf[I-1] <= ibuf[I]; // Shift one
        end
    end
end

```

```

        ibuf[IL] <= xiir;           // Input in register IL
    end
end

always @(posedge clk or posedge reset)
begin : GET                      // Get 4 samples at one time
    integer I;      // Loop variable
    if (reset) begin // Asynchronous clear
        for (I=0; I<=IL; I=I+1) x[I] <= 0;
    end else
        if (ena_out) begin
            for (I=0; I<=IL; I=I+1)
                x[I] <= ibuf[I]; // Take over input buffer
        end
    end

// Compute sum-of-products:
always @(posedge clk or posedge reset)
begin : SOP
// Matrix multiplier C-MOMS matrix:
//   x(0)      x(1)      x(2)      x(3)
//   0.3333    0.6667    0          0
//   -0.8333   0.6667   0.1667    0
//   0.6667   -1.5      1.0       -0.1667
//   -0.1667   0.5      -0.5      0.1667
    if (reset) begin // Asynchronous clear
        c0 <= 0; c1 <= 0; c2 <= 0; c3 <= 0;
        y0 <= 0; y1 <= 0; y2 <= 0; y3 <= 0;
        h0 <= 0; h1 <= 0; y <= 0;
    end else if (ena_out) begin
        c0 <= (85 * x[0] + 171 * x[1]) >>> 8;
        c1 <= (171 * x[1] - 213 * x[0] + 43 * x[2]) >>> 8;
        c2 <= (171 * x[0] - (43 * x[3])) >>> 8
                    - (3 * x[1] >>> 1) + x[2];
        c3 <= (43 * (x[3] - x[0])) >>> 8
                    + ((x[1] - x[2]) >>> 1);

// No Farrow structure, parallel LUT for delays
// for u=0:3, y=y+f(u)*d^u; end;
        y0 <= c0 * 256; // Use pipelined adder tree
        y1 <= c1 * d1[t];
        y2 <= c2 * d2[t];
        y3 <= c3 * d3[t];
        h0 <= y0 + y1;

```

```

    h1 <= y2 + y3;
    y  <= h0 + h1;
end
end

assign y_out = y >>> 8; // Connect to output
assign c0_o = c0; // Provide some test signals as outputs
assign c1_o = c1;
assign c2_o = c2;
assign c3_o = c3;
assign count_o = count;
assign ena_in_o = ena_in;
assign ena_out_o = ena_out;
assign xiir_o = xiir;

endmodule

```

```

//*****
// IEEE STD 1364-2001 Verilog file: db4latti.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module db4latti
  (input clk,                      // System clock
   input reset,                     // Asynchron reset
   output clk2,                     // Clock divider
   input signed [7:0] x_in,          // System input
   output signed [16:0] x_e, x_o,    // Even/odd x input
   output reg signed [8:0] g, h); // g/h filter output
// -----
  reg signed [7:0] x_wait;
  reg signed [16:0] sx_up, sx_low;
  reg clk_div2;
  wire signed [16:0] sxa0_up, sxa0_low;
  wire signed [16:0] up0, up1, low1;
  reg signed [16:0] low0;

  always @(posedge clk or posedge reset) // Split into even
  begin : Multiplex           // and odd samples at clk rate
    parameter even=0, odd=1;
    reg [0:0] state;

    if (reset) begin           // Asynchronous reset
      state <= even;
    end
    else begin
      if (state == even)
        state = odd;
      else
        state = even;
      end
      if (state == even)
        sx_low = up0;
      else
        sx_low = low1;
      end
      if (state == even)
        sx_up = sxa0_low;
      else
        sx_up = sxa0_up;
      end
    end
  end
endmodule

```

```

    sx_up <= 0; sx_low <= 0;
    clk_div2 <= 0; x_wait <= 0;
end else
case (state)
even : begin
// Multiply with 256*s=124
    sx_up <= (x_in <<< 7) - (x_in <<< 2);
    sx_low <= (x_wait <<< 7) - (x_wait <<< 2);
    clk_div2 <= 1;
    state <= odd;
end
odd : begin
    x_wait <= x_in;
    clk_div2 <= 0;
    state <= even;
end
endcase
end

//***** Multiply a[0] = 1.7321
// Compute: (2*sx_up - sx_up /4)-(sx_up /64 + sx_up /256)
assign sxa0_up = ((sx_up <<< 1) - (sx_up >>> 2))
            - ((sx_up >>> 6) + (sx_up >>> 8));
// Compute: (2*sx_low - sx_low/4)-(sx_low/64 + sx_low/256)
assign sxa0_low = ((sx_low <<< 1) - (sx_low >>> 2))
            - ((sx_low >>> 6) + (sx_low >>> 8));

//***** First stage -- FF in lower tree
assign up0 = sxa0_low + sx_up;
always @ (posedge clk or posedge reset)
begin: LowerTreeFF
if (reset) begin // Asynchronous clear
    low0 <= 0;
end else if (clk_div2)
    low0 <= sx_low - sxa0_up;
end

//***** Second stage: a[1]=-0.2679
// Compute: (up0 - low0/4) - (low0/64 + low0/256);
assign up1 = (up0 - (low0 >>> 2))
            - ((low0 >>> 6) + (low0 >>> 8));
// Compute: (low0 + up0/4) + (up0/64 + up0/256)
assign low1 = (low0 + (up0 >>> 2))
            + ((up0 >>> 6) + (up0 >>> 8));

```

```

assign x_e  = sx_up;           // Provide some extra
assign x_o  = sx_low;          // test signals
assign clk2 = clk_div2;

always @(posedge clk or posedge reset)
begin: OutputScale
    if (reset) begin           // Asynchronous clear
        g <= 0; h <= 0;
    end else if (clk_div2) begin
        g <= up1 >>> 8;      // i.e. up1 / 256
        h <= low1 >>> 8;     // i.e. low1 / 256;
    end
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: dwtden.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module dwtden
#(parameter D1L = 28, //D1 buffer length
          D2L = 10) // D2 buffer length
(input clk,                      // System clock
 input reset,                     // Asynchron reset
 input signed [15:0] x_in, // System input
 input signed [15:0] t4d1, t4d2, t4d3, t4a3,// Thresholds

 output signed [15:0] d1_out, // Level 1 detail
 output signed [15:0] a1_out, // Level 1 approximation
 output signed [15:0] d2_out, // Level 2 detail
 output signed [15:0] a2_out, // Level 2 approximation
 output signed [15:0] d3_out, // Level 3 detail
 output signed [15:0] a3_out, // Level 3 approximation
 // Debug signals:
 output signed [15:0] s3_out, a3up_out, d3up_out, // L3
 output signed [15:0] s2_out, s3up_out, d2up_out, // L2
 output signed [15:0] s1_out, s2up_out, d1up_out, // L1
 output signed [15:0] y_out); // System output
// -----
reg [2:0] count; // Cycle 2**max level
reg signed [15:0] x, xd; // Input delays
reg signed [15:0] a1, a2 ; // Analysis filter

```

```

wire signed [15:0] d1, d2, a3, d3 ; // Analysis filter
reg signed [15:0] d1t, d2t, d3t, a3t ;
                                         // Before thresholding
wire signed [15:0] abs_d1t, abs_d2t, abs_d3t, abs_a3t ;
                                         // Absolute values
reg signed [15:0] a1up, a3up, d3up ;
reg signed [15:0] a1upd, s3upd, a3upd, d3upd ;
reg signed [15:0] a1d, a2d; // Delay filter output
reg ena1, ena2, ena3; // Clock enables
reg t1, t2, t3; // Toggle flip-flops
reg signed [15:0] s2, s3up, s3, d2syn ;
reg signed [15:0] s1, s2up, s2upd ;
// Delay lines for d1 and d2
reg signed [15:0] d2upd [0:11];
reg signed [15:0] d1upd [0:29];

always @(posedge reset or posedge clk) // Control the
begin : FSM                         // system sample at clk rate
  if (reset) begin                  // Asynchronous reset
    count <= 0; ena1 <= 0; ena2 <= 0; ena3 <= 0;
  end else begin
    if (count == 7) count <= 0;
    else           count <= count + 1;
    case (count)   // Level 1 enable
      3'b001 : ena1 <= 1;
      3'b011 : ena1 <= 1;
      3'b101 : ena1 <= 1;
      3'b111 : ena1 <= 1;
      default : ena1 <= 0;
    endcase
    case (count) // Level 2 enable
      3'b001 : ena2 <= 1;
      3'b101 : ena2 <= 1;
      default : ena2 <= 0;
    endcase
    case (count) // Level 3 enable
      3'b101 : ena3 <= 1;
      default : ena3 <= 0;
    endcase
  end
end

// Haar analysis filter bank
always @(posedge reset or posedge clk)

```

```

begin : Analysis
  if (reset) begin          // Asynchronous clear
    x <= 0; xd <= 0; d1t <= 0; a1 <= 0; a1d <= 0;
    d2t <= 0; a2 <= 0; a2d <= 0; d3t <= 0; a3t <= 0;
  end else begin
    x <= x_in;
    xd <= x;
    if (ena1) begin // Level 1 analysis
      d1t <= x - xd;
      a1 <= x + xd;
      a1d <= a1;
    end
    if (ena2) begin // Level 2 analysis
      d2t <= a1 - a1d;
      a2 <= a1 + a1d;
      a2d <= a2;
    end
    if (ena3) begin // Level 3 analysis
      d3t <= a2 - a2d;
      a3t <= a2 + a2d;
    end
  end
end

// Take absolute values first
assign abs_d1t = (d1t>=0)? d1t : -d1t;
assign abs_d2t = (d2t>=0)? d2t : -d2t;
assign abs_d3t = (d3t>=0)? d3t : -d3t;
assign abs_a3t = (a3t>=0)? a3t : -a3t;

// Thresholding of d1, d2, d3 and a3
assign d1 = (abs_d1t > t4d1)? d1t : 0;
assign d2 = (abs_d2t > t4d2)? d2t : 0;
assign d3 = (abs_d3t > t4d3)? d3t : 0;
assign a3 = (abs_a3t > t4a3)? a3t : 0;

// Down followed by up sampling is implemented by setting
// every 2. value to zero
always @(posedge reset or posedge clk)
begin : Synthesis
  integer k;    // Loop variable
  if (reset) begin          // Asynchronous clear
    t1 <= 0; t2 <= 0; t3 <= 0;
    s3up <= 0;s3upd <= 0;

```

```

d3up <= 0; a3up <= 0; a3upd<=0; d3upd <= 0;
s3 <= 0; s2 <= 0;
s1 <= 0; s2up <= 0; s2upd <= 0;
for (k=0; k<=D2L+1; k=k+1) // delay to match s3up
    d2upd[k] <= 0;
for (k=0; k<=D1L+1; k=k+1) // delay to match s2up
    d1upd[k] <= 0;
end else begin
    t1 <= ~t1; // toggle FF level 1
    if (t1) begin
        d1upd[0] <= d1;
        s2up <= s2;
    end else begin
        d1upd[0] <= 0;
        s2up <= 0;
    end
    s2upd <= s2up;
    for (k=1; k<=D1L+1; k=k+1) // Delay to match s2up
        d1upd[k] <= d1upd[k-1];
    s1 <= (s2up + s2upd -d1upd[D1L] +d1upd[D1L+1]) >>> 1;
    if (ena1) begin
        t2 <= ~t2; // toggle FF level 2
        if (t2) begin
            d2upd[0] <= d2;
            s3up <= s3;
        end else begin
            d2upd[0] <= 0;
            s3up <= 0;
        end
        s3upd <= s3up;
        for (k=1; k<=D2L+1; k=k+1) // Delay to match s3up
            d2upd[k] <= d2upd[k-1];
        s2 <= (s3up +s3upd -d2upd[D2L] +d2upd[D2L+1])>>> 1;
    end
    if (ena2) begin // Synthesis level 3
        t3 <= ~t3; // toggle FF
        if (t3) begin
            d3up <= d3;
            a3up <= a3;
        end else begin
            d3up <= 0;
            a3up <= 0;
        end
    end

```

```

    a3upd <= a3up;
    d3upd <= d3up;
    s3 <= (a3up + a3upd - d3up + d3upd) >>> 1;
  end
end
end

assign a1_out = a1;// Provide some test signal as outputs
assign d1_out = d1;
assign a2_out = a2;
assign d2_out = d2;
assign a3_out = a3;
assign d3_out = d3;
assign a3up_out = a3up;
assign d3up_out = d3up;
assign s3_out = s3;
assign s3up_out = s3up;
assign d2up_out = d2upd[D2L];
assign s2_out = s2;
assign s1_out = s1;
assign s2up_out = s2up;
assign d1up_out = d1upd[D1L];
assign y_out = s1;
assign ena1_out = ena1;
assign ena2_out = ena2;
assign ena3_out = ena3;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: rader7.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module rader7          //---> Interface
  (input  clk,           // System clock
   input  reset,         // Asynchronous reset
   input [7:0] x_in,      // Real system input
   output reg signed[10:0] y_real, //Real system output
   output reg signed[10:0] y_imag); //Imaginary system output
// -----
  reg signed [10:0] accu;        // Signal for X[0]
// Direct bit access of 2D vector in Quartus Verilog 2001
// works; no auxiliary signal for this purpose necessary
  reg signed [18:0] im [0:5];
  reg signed [18:0] re [0:5];

```

```

// real is keyword in Verilog and can not be an identifier
// Tapped delay line array
reg signed [18:0] x57, x111, x160, x200, x231, x250 ;
// The filter coefficients
reg signed [18:0] x5, x25, x110, x125, x256;
// Auxiliary filter coefficients
reg signed [7:0] x, x_0; // Signals for x[0]
reg [1:0] state; // State variable
reg [4:0] count; // Clock cycle counter

always @(posedge clk or posedge reset) // State machine
begin : States // for RADER filter
    parameter Start=0, Load=1, Run=2;

    if (reset) begin // Asynchronous reset
        state <= Start; accu <= 0;
        count <= 0; y_real <= 0; y_imag <= 0;
    end else
        case (state)
            Start : begin // Initialization step
                state <= Load;
                count <= 1;
                x_0 <= x_in; // Save x[0]
                accu <= 0 ; // Reset accumulator for X[0]
                y_real <= 0;
                y_imag <= 0;
            end
            Load : begin // Apply x[5],x[4],x[6],x[2],x[3],x[1]
                if (count == 8) // Load phase done ?
                    state <= Run;
                else begin
                    state <= Load;
                    accu <= accu + x;
                end
                count <= count + 1;
            end
            Run : begin // Apply again x[5],x[4],x[6],x[2],x[3]
                if (count == 15) begin // Run phase done ?
                    y_real <= accu; // X[0]
                    y_imag <= 0; // Only re inputs => Im(X[0])=0
                    count <= 0; // Output of result
                    state <= Start; // and start again
                end else begin
                    y_real <= (re[0] >> 8) + x_0;
                end
            end
        endcase
    end
end

```

```

// i.e. re[0]/256+x[0]
y_imag <= (im[0] >> 8); // i.e. im[0]/256
state <= Run;
count <= count + 1;
end
end
endcase
end
// -----
always @(posedge clk or posedge reset) // Structure of the
begin : Structure // two FIR filters
integer k; // loop variable
if (reset) begin // in transposed form
  for (k=0; k<=5; k=k+1) begin
    re[k] <= 0; im[k] <= 0; // Asynchronous clear
  end
  x <= 0;
end else begin
  x <= x_in;
// Real part of FIR filter in transposed form
  re[0] <= re[1] + x160 ; // W^1
  re[1] <= re[2] - x231 ; // W^3
  re[2] <= re[3] - x57 ; // W^2
  re[3] <= re[4] + x160 ; // W^6
  re[4] <= re[5] - x231 ; // W^4
  re[5] <= -x57; // W^5

// Imaginary part of FIR filter in transposed form
  im[0] <= im[1] - x200 ; // W^1
  im[1] <= im[2] - x111 ; // W^3
  im[2] <= im[3] - x250 ; // W^2
  im[3] <= im[4] + x200 ; // W^6
  im[4] <= im[5] + x111 ; // W^4
  im[5] <= x250; // W^5
end
end
// -----
always @(posedge clk or posedge reset) // Note that all
begin : Coeffs // signals are globally defined
// Compute the filter coefficients and use FFs
if (reset) begin // Asynchronous clear
  x160 <= 0; x200 <= 0; x250 <= 0;
  x57 <= 0; x111 <= 0; x231 <= 0;
end else begin

```

```

        x160 <= x5 <<< 5;           // i.e. 160 = 5 * 32;
        x200 <= x25 <<< 3;           // i.e. 200 = 25 * 8;
        x250 <= x125 <<< 1;           // i.e. 250 = 125 * 2;
        x57  <= x25 + (x <<< 5); // i.e. 57 = 25 + 32;
        x111 <= x110 + x;            // i.e. 111 = 110 + 1;
        x231 <= x256 - x25;          // i.e. 231 = 256 - 25;
    end
end

always @*                         // Note that all signals
begin : Factors                  // are globally defined
// Compute the auxiliary factor for RAG without an FF
    x5   = (x <<< 2) + x; // i.e. 5 = 4 + 1;
    x25  = (x5 <<< 2) + x5; // i.e. 25 = 5*4 + 5;
    x110 = (x25 <<< 2) + (x5 <<< 2); // i.e. 110 = 25*4+5*4;
    x125 = (x25 <<< 2) + x25; // i.e. 125 = 25*4+25;
    x256 = x <<< 8;           // i.e. 256 = 2 ** 8;
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: fft256.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module fft256           //----> Interface
  (input  clk,           // System clock
   input  reset,          // Asynchronous reset
   input signed [15:0] xr_in, xi_in, // Real and imag. input
   output reg fft_valid, // FFT output is valid
   output reg signed [15:0] fftr, ffti, // Real/imag. output
   output [8:0] rcount_o, // Bitreverse index counter
   output [15:0] xr_out0, // Real first in reg. file
   output [15:0] xi_out0, // Imag. first in reg. file
   output [15:0] xr_out1, // Real second in reg. file
   output [15:0] xi_out1, // Imag. second in reg. file
   output [15:0] xr_out255, // Real last in reg. file
   output [15:0] xi_out255, // Imag. last in reg. file
   output [8:0] stage_o, gcount_o, // Stage and group count
   output [8:0] i1_o, i2_o, // (Dual) data index
   output [8:0] k1_o, k2_o, // Index offset
   output [8:0] w_o, dw_o, // Cos/Sin (increment) angle
   output reg [8:0] wo); // Decision tree location loop FSM
// -----

```



```

k2=N/2; dw = 1; fft_valid <= 0;
fftr <= 0; ffti <= 0; wo <= 0;
end else
  case (s)                                // Next State assignments
    start : begin
      s <= load; count <= 0; w <= 0;
      gcount = 0; stage= 1; i1 = 0; i2 = N/2; k1=N;
      k2=N/2; dw = 1; fft_valid <= 0; rcount <= 0;
    end
    load : begin      // Read in all data from I/O ports
      xr[count] <= xr_in; xi[count] <= xi_in;
      count <= count + 1;
      if (count == N)  s <= calc;
      else             s <= load;
    end
    calc : begin      // Do the butterfly computation
      tr = xr[i1] - xr[i2];
      xr[i1] <= xr[i1] + xr[i2];
      ti = xi[i1] - xi[i2];
      xi[i1] <= xi[i1] + xi[i2];
      cos_tr = cos * tr; sin_ti = sin * ti;
      xr[i2] <= (cos_tr >> 14) + (sin_ti >> 14);
      cos_ti = cos * ti; sin_tr = sin * tr;
      xi[i2] <= (cos_ti >> 14) - (sin_tr >> 14);
      s <= update;
    end
    update : begin      // all counters and pointers
      s <= calc;        // by default do next butterfly
      i1 = i1 + k1;     // next butterfly in group
      i2 = i1 + k2;
      wo <= 1;
      if ( i1 >= N-1 ) begin // all butterfly
        gcount = gcount + 1; // done in group?
        i1 = gcount;
        i2 = i1 + k2;
        wo <= 2;
        if ( gcount >= k2 ) begin // all groups done
          gcount = 0; i1 = 0; i2 = k2; // in stages?
          dw = dw * 2;
          stage = stage + 1;
          wo <= 3;
          if (stage > ldN) begin // all stages done
            s <= reverse;
            count = 0;
          end
        end
      end
    end
  endcase
end

```

```

        wo <= 4;
    end else begin // start new stage
        k1 = k2; k2 = k2/2;
        i1 = 0; i2 = k2;
        w <= 0;
        wo <= 5;
    end
end else begin // start new group
    i1 = gcount; i2 = i1 + k2;
    w <= w + dw;
    wo <= 6;
end
end
reverse : begin // Apply Bit Reverse
    fft_valid <= 1;
    for (k=0;k<=7;k=k+1) rcount[k] = count[7-k];
    fftr <= xr[rcount]; ffti <= xi[rcount];
    count = count + 1;
    if (count >= N) s <= done;
    else s <= reverse;
end
done : begin // Output of results
    s <= start; // start next cycle
end
endcase
end

assign xr_out0 = xr[0];
assign xi_out0 = xi[0];
assign xr_out1 = xr[1];
assign xi_out1 = xi[1];
assign xr_out255 = xr[255];
assign xi_out255 = xi[255];
assign i1_o = i1; // Provide some test signals as outputs
assign i2_o = i2;
assign stage_o = stage;
assign gcount_o = gcount;
assign k1_o = k1;
assign k2_o = k2;
assign w_o = w;
assign dw_o = dw;
assign rcount_o = rcount;
assign w_out = w;

```

```

    assign cos_out = cos;
    assign sin_out = sin;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: lfsr.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module lfsr          //----> Interface
  (input clk,           // System clock
   input reset,         // Asynchronous reset
   output [6:1] y);   // System output
// -----
  reg [6:1] ff; // Note that reg is keyword in Verilog and
                 // can not be variable name
  integer i; // loop variable

  always @(posedge clk or posedge reset)
  begin // Length 6 LFSR with xnor
    if (reset)           // Asynchronous clear
      ff <= 0;
    else begin
      ff[1] <= ff[5] ^~ ff[6];//Use non-blocking assignment
      for (i=6; i>=2 ; i=i-1)//Tapped delay line: shift one
        ff[i] <= ff[i-1];
    end
  end
  assign y = ff;           // Connect to I/O pins

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: lfsr6s3.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module lfsr6s3        //----> Interface
  (input clk,           // System clock
   input reset,         // Asynchronous reset
   output [6:1] y);   // System output
// -----
  reg [6:1] ff; // Note that reg is keyword in Verilog and

```

```

// can not be variable name

always @(posedge clk or posedge reset)
begin
    if (reset)           // Asynchronous clear
        ff <= 0;
    else begin           // Implement three-step
        ff[6] <= ff[3]; // length-6 LFSR with xnor;
        ff[5] <= ff[2]; // use non-blocking assignments
        ff[4] <= ff[1];
        ff[3] <= ff[5] ^~ ff[6];
        ff[2] <= ff[4] ^~ ff[5];
        ff[1] <= ff[3] ^~ ff[4];
    end
end

assign y = ff;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: ammod.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module ammod #(parameter W = 8) // Bit width - 1
    (input      clk,          // System clock
     input      reset,         // Asynchronous reset
     input signed [W:0] r_in,   // Radius input
     input signed [W:0] phi_in, // Phase input
     output reg signed [W:0] x_out, // x or real part output
     output reg signed [W:0] y_out, // y or imaginary part
     output reg signed [W:0] eps); // Error of results
// -----
    reg signed [W:0] x [0:3]; // There is bit access in 2D
    reg signed [W:0] y [0:3]; // array types in
    reg signed [W:0] z [0:3]; // Quartus Verilog 2001

always @(posedge clk or posedge reset)
begin: Pipeline
    integer k; // Loop variable
    if (reset) begin
        for (k=0; k<=3; k=k+1) begin // Asynchronous clear
            x[k] <= 0; y[k] <= 0; z[k] <= 0;
    end

```

```

    x_out <= 0; eps <= 0; y_out <= 0;
  end
else begin

  if (phi_in > 90) begin // Test for |phi_in| > 90
    x[0] <= 0;           // Rotate 90 degrees
    y[0] <= r_in;        // Input in register 0
    z[0] <= phi_in - 'sd90;
  end else if (phi_in < - 90) begin
    x[0] <= 0;
    y[0] <= - r_in;
    z[0] <= phi_in + 'sd90;
  end else begin
    x[0] <= r_in;
    y[0] <= 0;
    z[0] <= phi_in;
  end

  if (z[0] >= 0) begin           // Rotate 45 degrees
    x[1] <= x[0] - y[0];
    y[1] <= y[0] + x[0];
    z[1] <= z[0] - 'sd45;
  end else begin
    x[1] <= x[0] + y[0];
    y[1] <= y[0] - x[0];
    z[1] <= z[0] + 'sd45;
  end

  if (z[1] >= 0) begin           // Rotate 26 degrees
    x[2] <= x[1] - (y[1] >>> 1); // i.e. x[1] - y[1] /2
    y[2] <= y[1] + (x[1] >>> 1); // i.e. y[1] + x[1] /2
    z[2] <= z[1] - 'sd26;
  end else begin
    x[2] <= x[1] + (y[1] >>> 1); // i.e. x[1] + y[1] /2
    y[2] <= y[1] - (x[1] >>> 1); // i.e. y[1] - x[1] /2
    z[2] <= z[1] + 'sd26;
  end

  if (z[2] >= 0) begin           // Rotate 14 degrees
    x[3] <= x[2] - (y[2] >>> 2); // i.e. x[2] - y[2]/4
    y[3] <= y[2] + (x[2] >>> 2); // i.e. y[2] + x[2]/4
    z[3] <= z[2] - 'sd14;
  end else begin
    x[3] <= x[2] + (y[2] >>> 2); // i.e. x[2] + y[2]/4
  end
end

```

```

    y[3] <= y[2] - (x[2] >>> 2); // i.e. y[2] - x[2]/4
    z[3] <= z[2] + 'sd14;
end

x_out <= x[3];
eps <= z[3];
y_out <= y[3];
end
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: fir_lms.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// This is a generic LMS FIR filter generator
// It uses W1 bit data/coefficients bits
module fir_lms           //----> Interface
  #(parameter W1 = 8,      // Input bit width
         W2 = 16,     // Multiplier bit width 2*W1
         L = 2,       // Filter length
         Delay = 3) // Pipeline steps of multiplier
  (input clk,                // System clock
   input reset,              // Asynchronous reset
   input signed [W1-1:0] x_in, //System input
   input signed [W1-1:0] d_in, // Reference input
   output signed [W1-1:0] f0_out, // 1. filter coefficient
   output signed [W1-1:0] f1_out, // 2. filter coefficient
   output signed [W2-1:0] y_out, // System output
   output signed [W2-1:0] e_out); // Error signal
// -----
// Signed data types are supported in 2001
// Verilog, and used whenever possible
  reg signed [W1-1:0] x [0:1]; // Data array
  reg signed [W1-1:0] f [0:1]; // Coefficient array
  reg signed [W1-1:0] d;
  wire signed [W1-1:0] emu;
  reg signed [W2-1:0] p [0:1]; // 1. Product array
  reg signed [W2-1:0] xemu [0:1]; // 2. Product array
  wire signed [W2-1:0] y, sxt,y, e, sxt;
  wire signed [W2-1:0] sum; // Auxilary signals

  always @ (posedge clk or posedge reset)

```

```

begin: Store           // Store these data or coefficients
  if (reset) begin    // Asynchronous clear
    d <= 0; x[0] <= 0; x[1] <= 0; f[0] <= 0; f[1] <= 0;
  end else begin
    d <= d_in; // Store desired signal in register
    x[0] <= x_in; // Get one data sample at a time
    x[1] <= x[0]; // shift 1
    f[0] <= f[0] + xemu[0][15:8]; // implicit divide by 2
    f[1] <= f[1] + xemu[1][15:8];
  end
end

// Instantiate L multiplier
always @(*)
begin : MulGen1
  integer I; // loop variable
  for (I=0; I<L; I=I+1) p[I] <= x[I] * f[I];
end

assign y = p[0] + p[1]; // Compute ADF output

// Scale y by 128 because x is fraction
assign e = d - (y >>> 7);
assign emu = e >>> 1; // e*mu divide by 2 and
                        // 2 from xemu makes mu=1/4

// Instantiate L multipliers
always @(*)
begin : MulGen2
  integer I; // loop variable
  for (I=0; I<=L-1; I=I+1) xemu[I] <= emu * x[I];
end

assign y_out = y >>> 7; // Monitor some test signals
assign e_out = e;
assign f0_out = f[0];
assign f1_out = f[1];

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: fir4dlms.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****

```

```

// This is a generic DLMS FIR filter generator
// It uses W1 bit data/coefficients bits
module fir4dlms      //----> Interface
  #(parameter W1 = 8,    // Input bit width
        W2 = 16,   // Multiplier bit width 2*W1
        L = 2)    // Filter length
  (input clk,           // System clock
   input reset,         // Asynchronous reset
   input signed [W1-1:0] x_in,   //System input
   input signed [W1-1:0] d_in,   // Reference input
   output signed [W1-1:0] f0_out, // 1. filter coefficient
   output signed [W1-1:0] f1_out, // 2. filter coefficient
   output signed [W2-1:0] y_out,  // System output
   output signed [W2-1:0] e_out); // Error signal
// -----
// 2D array types memories are supported by Quartus II
// in Verilog, use therefore single vectors
reg signed [W1-1:0] x[0:4];
reg signed [W1-1:0] f[0:1];
reg signed [W1-1:0] d[0:2]; // Desired signal array
wire signed [W1-1:0] emu;
reg signed [W2-1:0] xemu[0:1]; // Product array
reg signed [W2-1:0] p[0:1];   // double bit width
reg signed [W2-1:0] y, e, sxt;
reg signed [W2-1:0] sxt;

always @ (posedge clk or posedge reset) // Store these data
begin: Store                         // or coefficients
  integer k;    // loop variable
  if (reset) begin                   // Asynchronous clear
    for (k=0; k<=2; k=k+1) d[k] <= 0;
    for (k=0; k<=4; k=k+1) x[k] <= 0;
    for (k=0; k<=1; k=k+1) f[k] <= 0;
  end else begin
    d[0] <= d_in; // Shift register for desired data
    d[1] <= d[0];
    d[2] <= d[1];
    x[0] <= x_in; // Shift register for data
    x[1] <= x[0];
    x[2] <= x[1];
    x[3] <= x[2];
    x[4] <= x[3];
    f[0] <= f[0] + xemu[0][15:8]; // implicit divide by 2
    f[1] <= f[1] + xemu[1][15:8];
  end
end

```

```

    end

    // Instantiate L pipelined multiplier
    always @(posedge clk or posedge reset)
    begin : Mul
        integer k, I;      // loop variable
        if (reset) begin      // Asynchronous clear
            for (k=0; k<=L-1; k=k+1) begin
                p[k] <= 0;
                xemu[k] <= 0;
            end
            y <= 0; e <= 0;
        end else begin
            for (I=0; I<L; I=I+1) begin
                p[I] <= x[I] * f[I];
                xemu[I] <= emu * x[I+3];
            end
            y <= p[0] + p[1]; // Compute ADF output
            // Scale y by 128 because x is fraction
            e <= d[2] - (y >>> 7);
        end
    end

    assign emu = e >>> 1; // e*mu divide by 2 and
                           // 2 from xemu makes mu=1/4

    assign y_out  = y >>> 7;    // Monitor some test signals
    assign e_out  = e;
    assign f0_out = f[0];
    assign f1_out = f[1];

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: pca.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module pca // -----> Interface
    (input clk,                      // System clock
     input reset,                     // Asynchron reset
     input signed [31:0] s1_in,       // 1. signal input
     input signed [31:0] s2_in,       // 2. signal input
     input signed [31:0] mu1_in,      // Learning rate 1. PC

```

```

input signed [31:0] mu2_in, // Learning rate 2. PC
output signed [31:0] x1_out, // Mixing 1. output
output signed [31:0] x2_out, // Mixing 2. output
output signed [31:0] w11_out, // Eigenvector [1,1]
output signed [31:0] w12_out, // Eigenvector [1,2]
output signed [31:0] w21_out, // Eigenvector [2,1]
output signed [31:0] w22_out, // Eigenvector [2,2]
output reg signed [31:0] y1_out, // 1. PC output
output reg signed [31:0] y2_out); // 2. PC output
// -----
// All data and coefficients are in 16.16 format
reg signed [31:0] s, s1, s2, x1, x2, w11, w12, w21, w22;
reg signed [31:0] h11, h12, y1, y2, mu1, mu2;
// Product double bit width
reg signed [63:0] a11s1, a12s2, a21s1, a22s2;
reg signed [63:0] x1w11, x2w12, w11y1, mu1y1, p12;
reg signed [63:0] x1w21, x2w22, w21y2, p21, w22y2;
reg signed [63:0] mu2y2, p22, p11;

wire signed [31:0] a11, a12, a21, a22, ini;
assign a11 = 32'h0000C000; // 0.75
assign a12 = 32'h00018000; // 1.5
assign a21 = 32'h00008000; // 0.5
assign a22 = 32'h00005555; // 0.333333
assign ini = 32'h00008000; // 0.5

always @(posedge clk or posedge reset)
begin : P1 // PCA using Sanger GHA
  if (reset) begin // reset/initialize all registers
    x1 <= 0; x2 <= 0; y1_out <= 0; y2_out <= 0;
    w11 <= ini; w12 <= ini; s1 <= 0; mu1 <= 0;
    w21 <= ini; w22 <= ini; s2 <= 0; mu2 <= 0;
  end else begin
    s1 <= s1_in; // place inputs in registers
    s2 <= s2_in;
    mu1 <= mu1_in;
    mu2 <= mu2_in;

    // Mixing matrix
    a11s1 = a11 * s1; a12s2 = a12 * s2;
    x1 <= (a11s1 >>> 16) + (a12s2 >>> 16);
    a21s1 = a21 * s1; a22s2 = a22 * s2;
    x2 <= (a21s1 >>> 16) - (a22s2 >>> 16);
  end
end

```

```

// first PC and eigenvector
x1w11 = x1 * w11;
x2w12 = x2 * w12;
y1 = (x1w11 >>> 16) + (x2w12 >>> 16);
w11y1 = w11 * y1;
mu1y1 = mu1 * y1;
h11 = w11y1 >>> 16;
p11 = (mu1y1 >>> 16) * (x1 - h11);
w11 <= w11 + (p11 >>> 16);

h12 = (w12 * y1) >>> 16;
p12 = (x2 - h12) * (mu1y1 >>> 16);
w12 <= w12 + (p12 >>> 16);

// second PC and eigenvector
x1w21 = x1 * w21; x2w22 = x2 * w22;
y2 = (x1w21 >>> 16) + (x2w22 >>> 16);
w21y2 = w21 * y2;
mu2y2 = mu2 * y2;
p21 = (mu2y2 >>> 16) * (x1 - h11 - (w21y2 >>> 16));
w21 <= w21 + (p21 >>> 16);

w22y2 = w22 * y2;
p22 = (mu2y2 >>> 16) * (x2 - h12 - (w22y2 >>> 16));
w22 <= w22 + (p22 >>> 16);
// registers y output
y1_out <= y1;
y2_out <= y2;
end
end

// Redefine bits as 32 bit SLV
assign x1_out = x1;
assign x2_out = x2;
assign w11_out = w11;
assign w12_out = w12;
assign w21_out = w21;
assign w22_out = w22;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: ica.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org

```

```

//*****
module ica // -----> Interface
  (input clk,           // System clock
   input reset,         // Asynchron reset
   input signed [31:0] s1_in, // 1. signal input
   input signed [31:0] s2_in, // 2. signal input
   input signed [31:0] mu_in, // Learning rate
   output signed [31:0] x1_out, // Mixing 1. output
   output signed [31:0] x2_out, // Mixing 2. output
   output signed [31:0] B11_out, // Demixing 1,1
   output signed [31:0] B12_out, // Demixing 1,2
   output signed [31:0] B21_out, // Demixing 2,1
   output signed [31:0] B22_out, // Demixing 2,2
   output reg signed [31:0] y1_out, // 1. component output
   output reg signed [31:0] y2_out); // 2. component output
// -----
// All data and coefficients are in 16.16 format

reg signed [31:0] s, s1, s2, x1, x2, B11, B12, B21, B22;
reg signed [31:0] y1, y2, f1, f2, H11, H12, H21, H22, mu;
reg signed [31:0] DB11, DB12, DB21, DB22;
// Product double bit width
reg signed [63:0] a11s1, a12s2, a21s1, a22s2;
reg signed [63:0] x1B11, x2B12, x1B21, x2B22;
reg signed [63:0] y1y1, y2f1, y1y2, y1f2, y2y2;
reg signed [63:0] muDB11, muDB12, muDB21, muDB22;
reg signed [63:0] B11H11, H12B21, B12H11, H12B22;
reg signed [63:0] B11H21, H22B21, B12H21, H22B22;

wire signed [31:0] a11, a12, a21, a22, one, negone;
assign a11 = 32'h0000C000; // 0.75
assign a12 = 32'h00018000; // 1.5
assign a21 = 32'h00008000; // 0.5
assign a22 = 32'h00005555; // 0.333333
assign one = 32'h00010000; // 1.0
assign negone = 32'hFFFF0000; // -1.0

always @(posedge clk or posedge reset)
begin : P1           // ICA using EASI
  if (reset) begin // reset/initialize all registers
    x1 <= 0; x2 <= 0; y1_out <= 0; y2_out <= 0;
    B11 <= one; B12 <= 0; s1 <= 0; mu <= 0;
    B21 <= 0; B22 <= one; s2 <= 0;
  end else begin

```

```

    s1 <= s1_in; // place inputs in registers
    s2 <= s2_in;
    mu <= mu_in;

    // Mixing matrix
    a11s1 = a11 * s1; a12s2 = a12 * s2;
    x1 <= (a11s1 >>> 16) + (a12s2 >>> 16);
    a21s1 = a21 * s1; a22s2 = a22 * s2;
    x2 <= (a21s1 >>> 16) - (a22s2 >>> 16);
    // New y values first
    x1B11 = x1 * B11; x2B12 = x2 * B12;
    y1 = (x1B11 >>> 16) + (x2B12 >>> 16);
    x1B21 = x1 * B21; x2B22 = x2 * B22;
    y2 = (x1B21 >>> 16) + (x2B22 >>> 16);
    // compute the H matrix
    // Build tanh approximation function for f1
    if (y1 > one) f1 = one;
    else if (y1 < negone) f1 = negone;
    else f1 = y1;
    // Build tanh approximation function for f2
    if (y2 > one) f2 = one;
    else if (y2 < negone) f2 = negone;
    else f2 = y2;

    y1y1 = y1 * y1;
    H11 = one - (y1y1 >>> 16) ;
    y2f1 = f1 * y2; y1y2 = y1 * y2; y1f2 = y1 * f2;
    H12 = (y2f1 >>> 16) - (y1y2 >>> 16) - (y1f2 >>> 16);
    H21 = (y1f2 >>> 16) - (y1y2 >>> 16) - (y2f1 >>> 16);
    y2y2 = y2 * y2;
    H22 = one - (y2y2 >>> 16);
    // update matrix Delta B
    B11H11 = B11 * H11; H12B21 = H12 * B21;
    DB11 = (B11H11 >>> 16) + (H12B21 >>> 16);
    B12H11 = B12 * H11; H12B22 = H12 * B22;
    DB12 = (B12H11 >>> 16) + (H12B22 >>> 16);
    B11H21 = B11 * H21; H22B21 = H22 * B21;
    DB21 = (B11H21 >>> 16) + (H22B21 >>> 16);
    B12H21 = B12 * H21; H22B22 = H22 * B22;
    DB22 = (B12H21 >>> 16) + (H22B22 >>> 16);
    // Store update matrix B in registers
    muDB11 = mu * DB11; muDB12 = mu * DB12;
    muDB21 = mu * DB21; muDB22 = mu * DB22;
    B11 <= B11 + (muDB11 >>> 16) ;

```

```

    B12 <= B12 + (muDB12 >>> 16);
    B21 <= B21 + (muDB21 >>> 16);
    B22 <= B22 + (muDB22 >>> 16);
    // register y output
    y1_out <= y1;
    y2_out <= y2;
end
end

assign x1_out = x1; // Redefine bits as 32 bit SLV
assign x2_out = x2;
assign B11_out = B11;
assign B12_out = B12;
assign B21_out = B21;
assign B22_out = B22;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: g711alaw.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// G711 includes A and mu-law coding for speech signals:
// A ~= 87.56; |x|<= 4095, i.e., 12 bit plus sign
// mu~=255; |x|<=8160, i.e., 14 bit
// -----
module g711alaw #(parameter WIDTH = 13) // Bit width
    (input clk, // System clock
     input reset, // Asynchron reset
     input signed [12:0] x_in, // System input
     output reg signed [7:0] enc, // Encoder output
     output reg signed [12:0] dec, // Decoder output
     output signed [13:0] err); // Error of results
// -----
    wire s;
    wire signed [12:0] x; // Auxiliary vectors
    wire signed [12:0] dif;
// -----
    assign s = x_in[WIDTH - 1]; // sign magnitude not 2C!
    assign x = {1'b0,x_in[WIDTH-2:0]};
    assign dif = dec - x_in; // Difference
    assign err = (dif>0)? dif : -dif; // Absolute error
// -----
    always @*

```

```

begin : Encode      // Mini floating-point format encoder
  if ((x>=0) && (x<=63))
    enc <= {s,2'b00,x[5:1]}; // segment 1
  else if ((x>=64) && (x<=127))
    enc <= {s,3'b010,x[5:2]}; // segment 2
  else if ((x>=128) && (x<=255))
    enc <= {s,3'b011,x[6:3]}; // segment 3
  else if ((x>=256) && (x<=511))
    enc <= {s,3'b100,x[7:4]}; // segment 4
  else if ((x>=512) && (x<=1023))
    enc <= {s,3'b101,x[8:5]}; // segment 5
  else if ((x>=1024) && (x<=2047))
    enc <= {s,3'b110,x[9:6]}; // segment 6
  else if ((x>=2048) && (x<=4095))
    enc <= {s,3'b111,x[10:7]}; // segment 7
  else enc <= {s,7'b0000000}; // + or - 0
end
// -----
always @*
begin : Decode // Mini floating point format decoder
  case (enc[6:4])
    3'b000 : dec <= {s,6'b000000,enc[4:0],1'b1};
    3'b010 : dec <= {s,6'b000001,enc[3:0],2'b10};
    3'b011 : dec <= {s,5'b00001,enc[3:0],3'b100};
    3'b100 : dec <= {s,4'b0001,enc[3:0],4'b1000};
    3'b101 : dec <= {s,3'b001,enc[3:0],5'b10000};
    3'b110 : dec <= {s,2'b01,enc[3:0],6'b100000};
    default : dec <= {s,1'b1,enc[3:0],7'b1000000};
  endcase
end
endmodule

//*****
// IEEE STD 1364-2001 Verilog file: adpcm.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
module adpcm                                //----> Interface
  (input clk,                               // System clock
   input reset,                             // Asynchron reset
   input signed [15:0] x_in,                // Input to encoder
   output [3:0] y_out,                      // 4 bit ADPCM coding word
   output signed [15:0] p_out,              // Predictor/decoder output
   output reg p_underflow, p_overflow);    // Predictor flags

```

```

    output signed [7:0] i_out,           // Index to table
    output reg i_underflow, i_overflow, // Index flags
    output signed [15:0] err,          // Error of system
    output [14:0] sz_out,             // Step size
    output s_out);                  // Sign bit
// -----
    reg signed [15:0] va, va_d; // Current signed adpcm input
    reg sign ; // Current adpcm sign bit
    reg [3:0] sdelta; // Current signed adpcm output
    reg [14:0] step; // Stepsize
    reg signed [15:0] sstep; // Stepsize including sign
    reg signed [15:0] valpred; // Predicted output value
    reg signed [7:0] index; // Current step change index

    reg signed [16:0] diff0, diff1, diff2, diff3;
                           // Difference val - valprev
    reg signed [16:0] p1, p2, p3;      // Next valpred
    reg signed [7:0] i1, i2, i3;       // Next index
    reg [3:0] delta2, delta3, delta4;
                           // Current absolute adpcm output
    reg [14:0] tStep;
    reg signed [15:0] vpdiff2, vpdiff3, vpdiff4 ;
                           // Current change to valpred

    // Quantization lookup table has 89 entries
    wire [14:0] t [0:88];

    // ADPCM step variation table
    wire signed [4:0] indexTable [0:15];

    assign indexTable[0]=-1; assign indexTable[1]=-1;
    assign indexTable[2]=-1; assign indexTable[3]=-1;
    assign indexTable[4]=2; assign indexTable[5]=4;
    assign indexTable[6]=6; assign indexTable[7]=8;
    assign indexTable[8]=-1; assign indexTable[9]=-1;
    assign indexTable[10]=-1; assign indexTable[11]=-1;
    assign indexTable[12]=2; assign indexTable[13]=4;
    assign indexTable[14]=6; assign indexTable[15]=8;
// -----
    assign t[0]=7; assign t[1]=8; assign t[2]=9;
    assign t[3]=10; assign t[4]=11; assign t[5]=12;
    assign t[6]= 13; assign t[7]= 14; assign t[8]= 16;
    assign t[9]= 17; assign t[10]= 19; assign t[11]= 21;
    assign t[12]= 23; assign t[13]= 25; assign t[14]= 28;

```

```

assign t[15]= 31; assign t[16]= 34; assign t[17]= 37;
assign t[18]= 41; assign t[19]= 45; assign t[20]= 50;
assign t[21]= 55; assign t[22]= 60; assign t[23]= 66;
assign t[24]= 73; assign t[25]= 80; assign t[26]= 88;
assign t[27]= 97; assign t[28]= 107; assign t[29]= 118;
assign t[30]= 130; assign t[31]= 143; assign t[32]= 157;
assign t[33]= 173; assign t[34]= 190; assign t[35]= 209;
assign t[36]= 230; assign t[37]= 253; assign t[38]= 279;
assign t[39]= 307; assign t[40]= 337; assign t[41]= 371;
assign t[42]= 408; assign t[43]= 449; assign t[44]= 494;
assign t[45]= 544; assign t[46]= 598; assign t[47]= 658;
assign t[48]= 724; assign t[49]= 796; assign t[50]= 876;
assign t[51]= 963; assign t[52]= 1060; assign t[53]= 1166;
assign t[54]= 1282; assign t[55]= 1411; assign t[56]= 1552;
assign t[57]= 1707; assign t[58]= 1878; assign t[59]= 2066;
assign t[60]= 2272; assign t[61]= 2499; assign t[62]= 2749;
assign t[63]= 3024; assign t[64]= 3327; assign t[65]= 3660;
assign t[66]= 4026; assign t[67]= 4428; assign t[68]= 4871;
assign t[69]= 5358; assign t[70]= 5894; assign t[71]= 6484;
assign t[72]= 7132; assign t[73]= 7845; assign t[74]= 8630;
assign t[75]= 9493; assign t[76]= 10442; assign t[77]= 11487;
assign t[78]= 12635; assign t[79]= 13899;
assign t[80]= 15289; assign t[81]= 16818;
assign t[82]= 18500; assign t[83]= 20350;
assign t[84]= 22385; assign t[85]= 24623;
assign t[86]= 27086; assign t[87]= 29794;
assign t[88]= 32767;
// -----
always @(posedge clk or posedge reset)
begin : Encode
  if (reset) begin // Asynchronous clear
    va <= 0; va_d <= 0;
    step <= 0; index <= 0;
    valpred <= 0;
  end else begin // Store in register
    va_d <= va; // Delay signal for error comparison
    va <= x_in;
    step <= t[i3];
    index <= i3;
    valpred <= p3; // Store predicted in register
  end
end

always @(va, va_d, step, index, valpred) begin

```

```
// ----- State 1: Compute difference from predicted sample
diff0 = va - valpred;
if (diff0 < 0) begin
    sign = 1;      // Set sign bit if negative
    diff1 = -diff0;// Use absolute value for quantization
end else begin
    sign = 0;
    diff1 = diff0;
end
// State 2: Quantize by devision and
// State 3: compute inverse quantization
// Compute: delta=floor(diff(k)*4./step(k)); and
// vpdiff(k)=floor((delta(k)+.5).*step(k)/4);
if (diff1 >= step) begin // bit 2
    delta2 = 4;
    diff2 = diff1 - step;
    vpdiff2 = step/8 + step;
end else begin
    delta2 = 0;
    diff2 = diff1;
    vpdiff2 = step/8;
end
if (diff2 >= step/2) begin //// bit3
    delta3 = delta2 + 2 ;
    diff3 = diff2 - step/2;
    vpdiff3 = vpdiff2 + step/2;
end else begin
    delta3 = delta2;
    diff3 = diff2;
    vpdiff3 = vpdiff2;
end
if (diff3 >= step/4) begin
    delta4 = delta3 + 1;
    vpdiff4 = vpdiff3 + step/4;
end else begin
    delta4 = delta3;
    vpdiff4 = vpdiff3;
end
// State 4: Adjust predicted sample based on inverse
if (sign)           // quantized
    p1 = valpred - vpdiff4;
else
    p1 = valpred + vpdiff4;
//----- State 5: Threshold to maximum and minimum -----
```

```

if (p1 > 32767) begin // Check for 16 bit range
    p2 = 32767; p_overflow <= 1;//2^15-1 two's complement
end else begin
    p2 = p1; p_overflow <= 0;
end
if (p2 < -32768) begin // -2^15
    p3 = -32768; p_underflow <= 1;
end else begin
    p3 = p2; p_underflow <= 0;
end

// State 6: Update the stepsize and index for stepsize LUT
i1 = index + indexTable[delta4];
if (i1 < 0) begin      // Check index range [0...88]
    i2 = 0; i_underflow <= 1;
end else begin
    i2 = i1; i_underflow <= 0;
end
if (i2 > 88) begin
    i3 = 88; i_overflow <= 1;
end else begin
    i3 = i2; i_overflow <= 0;
end
if (sign)
    sdelta = delta4 + 8;
else
    sdelta = delta4;
end

assign y_out  = sdelta;    // Monitor some test signals
assign p_out  = valpred;
assign i_out  = index;
assign sz_out = step;
assign s_out  = sign;
assign err = va_d - valpred;

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: reg_file.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// Description: This is a W x L bit register file.
module reg_file #(parameter W = 7, // Bit width - 1

```

```

N = 15) // Number of registers - 1
(input clk,           // System clock
 input reset,          // Asynchronous reset
 input reg_ena,         // Write enable active 1
 input [W:0] data,      // System input
 input [3:0] rd,        // Address for write
 input [3:0] rs,        // 1. read address
 input [3:0] rt,        // 2. read address
 output reg [W:0] s,    // 1. data
 output reg [W:0] t); // 2. data
// -----
// reg [W:0] r [0:N];

always @(posedge clk or posedge reset)
begin : MUX           // Input mux inferring registers
  integer k;           // loop variable
  if (reset)           // Asynchronous clear
    for (k=0; k<=N; k=k+1) r[k] <= 0;
  else if ((reg_ena == 1) && (rd > 0))
    r[rd] <= data;
end

// 2 output demux without registers
always @*
begin : DEMUX
  if (rs > 0) // First source
    s = r[rs];
  else
    s = 0;
  if (rt > 0) // Second source
    t = r[rt];
  else
    t = 0;
end

endmodule

//*****
// IEEE STD 1364-2001 Verilog file: trisc0.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****
// Title: T-RISC stack machine
// Description: This is the top control path/FSM of the
// T-RISC, with a single 3 phase clock cycle design

```

```

// It has a stack machine/0-address type instruction word
// The stack has only 4 words.
// -----
module trisc0 #(parameter WA = 7, // Address bit width -1
                  WD = 7) // Data bit width -1
    (input  clk,           // System clock
     input  reset,         // Asynchronous reset
     output jc_out,        // Jump condition flag
     output me_ena,        // Memory enable
     input [WD:0] iport,    // Input port
     output reg [WD:0] oport, // Output port
     output signed [WD:0] s0_out, // Stack register 0
     output signed [WD:0] s1_out, // Stack register 1
     output [WD:0] dmd_in,   // Data memory data read
     output [WD:0] dmd_out,  // Data memory data read
     output [WD:0] pc_out,   // Progamm counter
     output [WD:0] dma_out,  // Data memory address write
     output [WD:0] dma_in,   // Data memory address read
     output [7:0] ir_imm,    // Immidiate value
     output [3:0] op_code); // Operation code
// -----
//parameter ifetch=0, load=1, store=2, incpc=3;
reg [1:0] state;

wire [3:0] op;
wire [WD:0] imm, dmd;
reg signed [WD:0] s0, s1, s2, s3;
reg [WA:0] pc;
wire [WA:0] dma;
wire [11:0] pmd, ir;
wire eq, ne, not_clk;
reg mem_ena, jc;

// OP Code of instructions:
parameter
add = 0, neg = 1, sub = 2, opand = 3, opor = 4,
inv = 5, mul = 6, pop = 7, pushi = 8, push = 9,
scan = 10, print = 11, cne = 12, ceq = 13, cjp = 14,
jmp = 15;

always @(*) // sequential FSM of processor
            // Check store in register ?
    case (op) // always store except Branch
        pop : mem_ena <= 1;

```

```

    default : mem_ena <= 0;
endcase

always @(negedge clk or posedge reset)
if (reset == 1) // update the program counter
    pc <= 0;
else begin // use falling edge
    if ((op==cjp) & (jc==0)) | (op==jmp))
        pc <= imm;
    else
        pc <= pc + 1;
end

always @(posedge clk or posedge reset)
if (reset) // compute jump flag and store in FF
    jc <= 0;
else
    jc <= ((op == ceq) & (s0 == s1)) |
                    ((op == cne) & (s0 != s1));

// Mapping of the instruction, i.e., decode instruction
assign op = ir[11:8]; // Operation code
assign dma = ir[7:0]; // Data memory address
assign imm = ir[7:0]; // Immidiate operand

prog_rom brom
( .clk(clk), .reset(reset), .address(pc), .q(pmd));
assign ir = pmd;

assign not_clk = ~clk;

data_ram bram
( .clk(not_clk), .address(dma), .q(dmd),
  .data(s0), .we(mem_ena));

always @(posedge clk or posedge reset)
begin : P3
    integer temp;
    if (reset) begin // Asynchronous clear
        s0 <= 0; s1 <= 0; s2 <= 0; s3 <= 0;
        oport <= 0;
    end else begin
        case (op)
            add      : s0 <= s0 + s1;

```

```

    neg      : s0  <= -s0;
    sub      : s0  <= s1 - s0;
    opand   : s0  <= s0 & s1;
    opor    : s0  <= s0 | s1;
    inv     : s0  <= ~ s0;
    mul     : begin temp = s0 * s1; // double width
              s0 <= temp[WD:0]; end // product
    pop     : s0  <= s1;
    push    : s0  <= dmd;
    pushi   : s0  <= imm;
    scan    : s0 <= iport;
    print   : begin oport <= s0; s0<=s1; end
    default: s0 <= 0;
endcase
case (op) // Specify the stack operations
  pushi, push, scan : begin s3<=s2;
    s2<=s1; s1<=s0; end           // Push type
  cjp, jmp, inv | neg : ;        // Do nothing for branch
  default : begin s1<=s2; s2<=s3; s3<=0; end
                // Pop all others
endcase
end
end

// Extra test pins:
assign dmd_out = dmd; assign dma_out = dma; //Data memory
assign dma_in = dma; assign dmd_in = s0;
assign pc_out = pc; assign ir_imm = imm;
assign op_code = op; // Program control
// Control signals:
assign jc_out = jc; assign me_ena = mem_ena;
// Two top stack elements:
assign s0_out = s0; assign s1_out = s1;

endmodule

```

Appendix B. Design Examples Synthesis Results

The synthesis results for all examples can be easily reproduced for the Quartus version installed on your computer by using the script `qvhd1.tcl` for VHDL or `qv.tcl` for Verilog available in the source code directories of the CD-ROM. Run the VHDL TCL script with

```
quartus_sh -t qvhdl.tcl
```

to compile all designs. The next step is to run the resource and timing analysis with

```
quartus_sta -t fmax4all.tcl
```

The script produces four parameters for each design. For the `trisc0.vhd`, for instance, we get the following:

```
....  
-----  
trisc0 (Clock clk) : Fmax = 92.66 (Restricted Fmax = 92.66)  
trisc0 LEs: 171 / 114,480 (< 1 % )  
trisc0 M9K bits: 256 / 3,981,312 (< 1 % )  
trisc0 9-bit DSP blocks: 1 / 532 (< 1 % )  
-----  
....
```

then use a utility like `grep` through the report `qvhd1.txt` file using `Fmax`, `LEs`: etc.

From the script you will notice that the following special options of Quartus II web edition 12.1 were used:

- Device set Family to Cyclone IV E and then under Available devices select EP4CE115F29C7.
- For Timing Analysis Settings set Default required fmax: to 1 ns.
- For Analysis & Synthesis Settings from the Assignments menu
 - set Optimization Technique to Speed
 - Deselect Power-Up Don't Care
- In the Fitter Settings select as Fitter effort Standard Fit (highest effort)

The table below displays the results for all VHDL and Verilog examples given in this book. The table is structured as follows. The first column shows the entity or module name of the design. Columns 2 to 6 are data for the VHDL designs: the number of LEs shown in the report file; the number of 9×9 -bit multipliers; the number of M9K memory blocks; the registered performance F_{max} using the TimeQuest slow 85C model; and the page with the source code. The same data are provided for the Verilog design examples, shown in columns 7 to 9. Note that VHDL and Verilog produce the same data for a number of 9×9 -bit multipliers most of the time, except for the four designs **ica** (Verilog 184 multiplier), **pca** (Verilog 138 multiplier), **iir5para** (Verilog 58 multiplier), and **iir51wdf** (Verilog 18 multiplier). LEs and registered performance never match. The number of M9K memory blocks do not match for the three designs **fft256**, **fun_text**, and **trisc0**. In Verilog the ROM LUTs are synthesized to M9K blocks, while in VHDL LEs are used. An **fpu** design is not available in Verilog. A few designs don't use registers and a registered performance cannot be measured.

Design	LEs	9 × 9 Mult. vhd/v	VHDL			LEs	Verilog	
			M9Ks	f_{MAX} MHz	Page		f_{MAX} MHz	Page
add1p	125	0	0	350.63	83	77	336.25	797
add2p	233	0	0	243.43	83	143	318.17	798
add3p	372	0	0	231.43	83	228	278.47	799
adpcm	531	0	0	49.5	618	510	56.0	870
ammod	264	0	0	197.39	512	222	298.78	859
arctan	106	3	0	32.71	145	105	33.05	806
cic3r32	341	0	0	282.49	321	339	280.11	831
cic3s32	209	0	0	290.02	330	206	294.2	833
cmoms	549	3	0	95.27	369	421	102.81	841
cmul7p8	48	0	0	-	63	48	-	797
cordic	276	0	0	209.6	137	172	317.97	804
dapara	39	0	0	205.17	212	39	205.17	819
dasign	52	0	0	258.4	204	39	331.56	817
db4latti	420	0	0	58.11	390	248	99.02	845
db4poly	167	0	0	618.43	310	156	554.32	829
div_aegp	45	4	0	124.91	103	44	129.28	801
div_res	106	0	0	263.5	96	89	269.25	803
dwtden	879	0	1	120.93	406	889	164.28	847
example	33	0	0	267.24	17	32	457.67	795
farrow	363	3	0	39.82	358	268	58.25	839
fft256	34,340	8	0/2	31.12	442	33,926	31.16	854
fir4dlms	106	4	0	261.57	568	105	260.62	862
fir_gen	93	4	0	157.38	182	93	153.66	813
fir_lms	51	4	0	69.26	561	51	70.2	861
fir_srg	109	0	0	88.35	193	79	99.81	814
fpu	8112	7	0	-	120	-	-	-
fun_text	180	0	0/1	250.63	33	32	306.65	796
g711alaw	70	0	0	-	358	97	-	869
ica	2275	172/184	0	17.87	605	2091	17.84	866
iir	62	0	0	147.3	227	30	224.82	820
iir5lwdf	764	12/18	0	55.97	296	611	52.46	828
iir5para	624	51/58	0	87.69	267	513	86.72	825
iir5sfix	2474	128	0	46.99	255	2474	47.08	824
iir_par	236	0	0	479.39	247	185	430.29	822
iir_pipe	123	0	0	215.05	241	75	350.14	821
lfsr	6	0	0	944.29	495	6	944.29	858
lfsr6s3	6	0	0	931.97	497	6	931.97	858
ln	88	10	0	29.2	156	88	29.2	807
magnitude	96	0	0	119.59	167	145	107.34	812
pca	2447	180/138	0	18.46	596	1609	23.82	864
rader7	428	0	0	138.45	429	403	151.56	851
rc_sinc	880	0	0	59.53	350	847	78.52	835
reg_file	226	0	0	-	653	226	-	874
sqrt	261	2	0	86.23	161	244	112.1	809
trisc0	171	1	1/2	92.66	701	140	85.59	875

Appendix C. VHDL and Verilog Coding Keywords

Unfortunately, today we find *two* HDL languages are popular. The US west coast and Asia prefer Verilog, while the US east coast and Europe more frequently use VHDL. For digital signal processing with FPGAs, both languages seem to be well suited, but some VHDL examples were in the past a little easier to read because of the supported signed arithmetic and multiply/divide operations in the IEEE VHDL 1076-1987 and 1076-1993 standards. This gap has disappeared with the introduction of the Verilog IEEE standard 1364-2001, as it also includes signed arithmetic. Other constraints may include personal preferences, EDA library and tool availability, data types, readability, capability, and language extensions using PLIs, as well as commercial, business and marketing issues, to name just a few. A detailed comparison can be found in the book by Smith [3]. Tool providers acknowledge today that both languages need to be supported.

It is therefore a good idea to use an HDL code style that can easily be translated into either language. An important rule is to avoid any keyword in *both* languages in the HDL code when naming variables, labels, constants, user types, etc. The IEEE standard VHDL 1076-1993 uses 97 keywords (see VHDL 1076-1993 Language Reference Manual (LRM) on p. 179) and an extra 18 keywords are used in VHDL 1076-2008 (see VHDL 1076-2008 Language Reference Manual (LRM) on p. 236). New in VHDL 1076-2008 are:

```
ASSUME, ASSUME_GUARANTEE, CONTEXT, COVER, DEFAULT, FAIRNESS,  
FORCE, PARAMETER, PROPERTY, PROTECTED, RELEASE, RESTRICT,  
RESTRICT_GUARANTEE, SEQUENCE, STRONG, VMODE, VPROP, VUNIT
```

which are in version Quartus 12.1 *not* highlighted in blue in the editor but may be recognized in later Quartus versions. The IEEE standard Verilog 1364-1995, on the other hand, has 102 keywords (see LRM, p. 604). New in Verilog 1076-2001 are:

```
automatic, cell, config, design, endconfig, endgenerate,  
generate, genvar, incdir, include, instance, liblist,  
library, localparam, noshowcancelled, pulsetstyle_onevent,  
pulsetstyle_ondetect, showcancelled, signed, unsigned, use
```

Together, both HDL languages (Verilog 1364-2001 and VHDL 1076-2008) have 215 keywords, including 23 in common. The Table below shows VHDL

1076-2008 keywords in capital letters, Verilog 1364-2001 keywords in small letters, and the common keywords with a capital first letter.

Table Appendix C:1. VHDL 1076-1993 and Verilog 1364-2001 keywords

```

ABS ACCESS AFTER ALIAS ALL always And ARCHITECTURE ARRAY ASSERT
assign ASSUME ASSUME_GUARANTEE ATTRIBUTE automatic Begin BLOCK
BODY buf BUFFER bufif0 bufif1 BUS Case casex casez cell cmos
config COMPONENT CONFIGURATION CONSTANT CONTEXT COVER deassign
Default defparam design disable DISCONNECT DOWNT0 edge Else
ELSIF End endcase endconfig endfunction endgenerate endmodule
endprimitive endspecify endtable endtask ENTITY event EXIT
FAIRNESS FILE For Force forever fork Function Generate GENERIC
genvar GROUP GUARDED highz0 highz1 If ifnone IMPURE IN incdir
include INERTIAL initial Inout input instance integer IS join
LABEL large liblist Library LINKAGE LITERAL LOOP localparam
macromodule MAP medium MOD module Nand negedge NEW NEXT nmos Nor
noshowcancelled Not notif0 notif1 NULL OF ON OPEN Or OTHERS OUT
output PACKAGE Parameter pmos PORT posedge POSTPONED primitive
PROCEDURE PROCESS PROPERTY PROTECTED pullo pull1 pulldown pullup
pulsestyle_onevent pulsestyle_onedetect PURE RANGE rcmos real
realtime RECORD reg REGISTER REJECT Release REM repeat REPORT
RESTRICT RESTRICT_GUARANTEE RETURN rnmos ROL ROR rpmos rtran
rtranif0 rtranif1 scalared SELECT SEQUENCE SEVERITY SHARED
showcancelled SIGNAL signed OF SLA SLL small specify specparam
SRA SRL STRONG strong0 strong1 SUBTYPE supply0 supply1 table task
THEN time TO tran tranif0 tranif1 TRANSPORT tri tri0 tri1 triand
trior trireg TYPE UNAFFECTED UNITS unsigned UNTIL Use VARIABLE
VMODE VPROP VUNIT vectored Wait wand weak0 weak1 WHEN While wire
WITH wor Xnor Xor

```

Appendix D. CD-ROM Content

The accompanying CD-ROM includes:

- All VHDL/Verilog design examples and scripts to compile
- Utility programs and files

To install the Quartus II 12.1 web edition software first go to Altera's webpage www.altera.com and click on **Design Tools & Services** and select **Design Software**. Select the web edition unless you have a full subscription. Download the software including the free ModelSim-Altera package.

Altera frequently update the Quartus II software to support new devices and other improvements and you may consider downloading the latest Quartus II version from the Altera webpage directly, but keep in mind that the files are large and that the synthesis results will differ slightly for another version than 12.1 used for the book.

The design examples for the book are located in the directories **vhdl** and **verilog** for the VHDL and Verilog examples, respectively. These directories contain, for each example, the following files:

- The VHDL or Verilog source code (*.vhd and *.v)
- The Quartus project files (*.qpf)
- The Quartus setting files (*.qsf)
- The ModelSim simulator stimuli script (*.do)
- The files for timing simulation (*.vho and *.vo)

To simplify the compilation and postprocessing, the source code directories include the additional (*.bat) files and Tcl scripts shown below:

File	Comment
qvhdl.tcl or qv.tcl	Tcl script to compile all design examples. Note that the device can be changed from Cyclone IV to Flex, Apex or Stratix just by changing the comment sign # in column 1 of the script.
fmax4all.bat	Script to compute the used resources and the maximum performance of the designs.
qclean.bat	Cleans all temporary Quartus II compiler files, but not the report files (*.map.rpt), and the project files *.qpf and *.qsf.
qveryclean.bat	Cleans all temporary compiler files, <i>including</i> all report files (*.rpt) and project files.

Use the DOS prompt and type

```
quartus_sh -t qvhdl.tcl
```

to compile all design examples and then

```
quartus_sta -t fmax4all.tcl
```

to determine the performance and resources. Then run the qclean.bat to remove the unnecessary files. The Tcl script language developed by the Berkeley Professor John Ousterhout [442–444] (used by most modern CAD tools: Altera Quartus, Xilinx ISE, ModelTech, etc.) allows a comfortable scripting language to define setting, specify functions, etc. Given the fact that many tools also use the graphic toolbox Tcl/Tk we have witnessed that many tools now also look almost the same.

The script includes all settings and also alternative device definitions. The script produces four parameters for each design. For the trisc0.vhd, for instance, we get:

```
....  
-----  
trisc0 (Clock clk) : Fmax = 92.66 (Restricted Fmax = 92.66)  
trisc0 LEs: 171 / 114,480 ( < 1 % )  
trisc0 M9K bits: 256 / 3,981,312 ( < 1 % )  
trisc0 9-bit DSP blocks: 1 / 532 ( < 1 % )  
-----  
....
```

The results for all examples are summarized in Appendix B Table p. 881.

Other devices are specified in the script and include:

- EPF10K20RC240-4 from the UP1 University board
- EPF10K70RC240-4 from the UP2 University board
- EP20K200EFC484-2X from the Nios development board

- EP2C35F672C6 from the DE2 University board
- EP4SGX230 from the DE4 University board
- EP1S10F484C5, EP1S25F780C5, and EP2S60F1020C4ES from other DSP boards available from Altera

For the simulation stimuli *.do files are provided for the ModelSim simulator that are almost identical for the VHDL and Verilog projects. A simulation file example is shown in Chap. 1, p. 38. Both use a `tb_ini.do` initialization file that compiles the source code and provides an `add_local` function that allows one to add additional signals in the functional simulation that may not be available in timing simulation. For the `fun_text` project, for instance, we use `do fun_text.do 0` for RTL and `do fun_text.do 1` for timing simulation in the ModelSim simulator transcript window. Timing simulation requires a full compilation first and the output files `*.vo` or `*.vho` must be placed in the source code directory.

Utility Programs and Files

A couple of extra utility programs are also included on the CD-ROM¹ and can be found in the directory `util`:

File	Description
<code>sine.exe</code>	Program to generate the VHDL and Verilog sine for the function generator in Chap. 1
<code>csd.exe</code>	Program to find the canonical signed digit representation of integers or fractions as used in Chap. 2
<code>fp_ops.exe</code>	Program to compute the floating-point test data used in Chap. 2
<code>dagen.exe</code>	Program to generate the VHDL code for the distributed arithmetic files used in Chap. 3
<code>ragopt.exe</code>	Program to compute the reduced adder graph for constant-coefficient filters as used in Chap. 3. It has ten predefined lowpass and half-band filters. The program uses a MAG cost table stored in the file <code>mag14.dat</code>
<code>cic.exe</code>	Program to compute the parameters for a CIC filter as used in Chap. 5

The programs are compiled using the author's MS Visual C++ standard edition software (available for \$50–100 at all major retailers) for DOS window

¹ You need to copy the programs to your hard-drive or memory stick first; you cannot start them from the CD directly since the programs write out the results in text files.

applications and should therefore run on Windows 95 or higher. The DOS script **Testall.bat** produces the examples used in the book.

Also under **util** we find the following utility files:

File	Description
quickver.pdf	Quick reference card for Verilog HDL from QUALIS
quickvhdl.pdf	Quick reference card for VHDL from QUALIS
quicklog.pdf	Quick reference card for the IEEE 1164 logic package from QUALIS
93vhdl.vhd	The IEEE VHDL 1076-1993 keywords
2008vhdl.vhd	The IEEE VHDL 1076-2008 keywords
95key.v	The IEEE Verilog 1364-1995 keywords
01key.v	The IEEE Verilog 1364-2001 keywords
95direct.v	The IEEE Verilog 1364-1995 compiler directives
95tasks.v	The IEEE Verilog 1364-1995 system tasks and functions

In addition, the CD-ROM includes a collection of useful Internet links (see file **dsp4fpga.htm** under **util**), such as device vendors, software tools, VHDL and Verilog resources, and links to online available HDL introductions, e.g., the “Verilog Handbook” by Dr. D. Hyde and “The VHDL Handbook Cookbook” by Dr. P. Ashenden.

(L)WDF Filter Toolbox

The (L)WDF toolbox written by Lincklaen Arriens can be found in the **lwdf** folder. There are also two PDF manuals that help you get started:

- **WDF_toolbox_UG_v1_0.pdf** is the (L)WDF Toolbox MATLAB Users Guide that includes tutorial to design the (L)WDF filters
- **WDF_toolbox_RG_v1_0.pdf** is the (L)WDF Toolbox MATLAB reference guide that includes a description of the available functions

These files are used in Chap. 4 to design WDF and LWDF narrow band IIR filters.

Compressed Sound Data

Under **sound** we find the following speech data files used in Chap. 8:

File	Description
Speech_PCM16bit.wav	Original speech data in 16-bit precision (no compression)
Speech_PCM8bit.wav	Original speech data in 8-bit precision (no compression)
Speech_PCM4bit.wav	Original speech data in 4-bit precision (no compression)
Speech_A_LAW8bit.wav	Compressed speech data using 8 bits per sample A-law compression
Speech_PCM4Lloyd.wav	Compressed speech data using 4 bits per sample Lloyd optimal quantizer
Speech_ADPCM4bit.wav	Compressed speech data using 4 bits per sample ADPCM method

Micropocessor Project Files and Programs

All microprocessor-related tools and documents can be found in the uP folder. Six software Flex/Bison projects along with their compiler scripts are included:

- **build1.bat** and **simple.l** are used for a simple Flex example.
- **build2.bat**, **d_ff.vhd**, and **vhdlcheck.l** are a basic VHDL lexical analysis.
- **build3.bat**, **asm2mif.l**, and **add2.txt** are used for a simple Flex example.
- **build4.bat**, **add2.y**, and **add2.txt** are used for a simple Bison example.
- **build5.bat**, **calc.l**, **calc.y**, and **calc.txt** are infix calculators and are used to demonstrate the Bison/Flex communication.
- **build6.bat**, **c2asm.h**, **c2asm.h**, **c2asm.c**, **lc2asm.c**, **yc2asm.c**, and **factorial.c** are used for a C-to-assembler compiler for a stack computer.

The ***.txt** files are used as input files for the programs. The **buildx.bat** can be used to compile each project separately; alternatively you can use the **uPrunall.bat** under Unix to compile and run all files in one step. The compiled files that run under SunOS UNIX end with ***.exe** while the DOS programs end with ***.com**.

Here is a short description of the other supporting files in the uP directory: **Bison.pdf** contains the Bison compiler, i.e., the YACC-compatible parser generator, written by Charles Donnelly and Richard Stallman; **Flex.pdf** is the description of the fast scanner generator written by Vern Paxson.

VGA Project Files

To get started with the VGA project, connect the DE2 board to your PC via USB cable, add a power supply for the board and the VGA cable to a VGA monitor. After you turn on the power you should see the test picture of the DE2 board on the VGA monitor. This is the startup configuration that is factory programmed in the DE2's E²ROM. Now copy the whole directory `DE2_115_ImageVGA` from the CD to your PC, memory stick, or network drive, and you are ready to start the Quartus software and load the project or double click `DE2_115_ImageVGA.qpf`. You can download the project `DE2_115_ImageVGA.sof` to the board to become familiar with the project and the switches. Observe the VGA display, LCD and the eight seven segment displays. Try to change the MSB and the LSB of the slider switches and observe the changes in the VGA display.

If you want to modify the project here is a brief description of the major files of the projects:

- `DE2_115_ImageVGA.v` is the top level design file that includes the edge detection filter, instantiation of the image memory, and connections to the I/O pins. Since the project does not use the Nios II processor no `Qsys` file is required. All design files including the driver for the I/O are already included in the source code directory.
- `VGA_wave.do` is the script to run the `ModelSim` simulation. Remember that we use the timing simulation since the loading of the large MIF file in functional simulation has excessive memory requirements. Make a full compile first before you start the simulation. Start `ModelSim` and then run the script with `do VGA_wave.do 1`. The parameter “1” means timing simulation.
- If you like to test other images you need to do two things. First you need to convert your 640×480 BMP image using the MATLAB script `bm2txt.m` or the factory provided `PrintNum.exe` to an MIF file. You can find these files in the `VGA_DATA` directory of the project. There are also several other test picture you can try. For the second step, in Quartus start the MegaWizard, load/edit the megafunction file `img_data.v` and browse to the new MIF file. Then recompile the whole project and download the SOF file to the board.
- `qclean.bat` cleans the temporary files, but not the SOF file. In addition you should delete the `db` directory before moving the project to another location.

After you make any changes to the project make sure to recompile the whole project before downloading the SOF file to the board. This project has no `Qsys` files and a Nios II system generation is therefore unnecessary.

Image Processing Project Files

To get started with the median filter Nios II software project, connect the DE2 to your PC via USB cable, add a power supply for the board and the VGA cable to a VGA monitor. After you turn on the power you should see the test picture of the DE2 board on the VGA monitor. This is the startup configuration that is factory programmed in the DE2's E²ROM. Now copy the whole directory `DE2_115_ImageProcessing` from the CD to your PC, memory stick, or network drive, and you are ready to start the Quartus software and load the project or double click `DE2_115_ImageProcessing.qpf`. You should then download the project `DE2_115_ImageProcessing.sof` to the board. The VGA display will show random black-and-white pixels. Next start the **Nios II Software Build Tools for Eclipse**. You may try to import the whole software project from the software/median folder; however we found that most often pathes or files could not be found in the right location. It is usually more successful if you start with a "Hello World" project and then copy the required files in this hello world project. Therefore, we recommend that within Eclipse you generate a "Hello World" project. If you run it as Nios hardware it will give out in the terminal window the message "Hello from Nios II!" After you have successfully run this program replace the `hello_world.c` file with the `median.c` file and copy the file `Qpicture.mif` into the same directory in which you have the hello world project files. You find the required new source files in the `c-source` subdirectory. Now you need to enable the host file system support. Right click `median_bsp` in **Project Explorer** and Start then **Nios II → BSP Editor** In **BSP Editor** select **Software Packages** and Enable `altera_hostfs`. Finally click the **Generate** button. Then right click the project in the **Project Explore** window and select **Debug As → Nios II Hardware**. Then the project will be downloaded to the board and the debug window opens. Then press the **Run** button or F8. The image is transferred to the board, noise is added, and horizontal and vertical filters are applied. The LEDs indicate the status of the steps. The slider switches (SWs) are used as follows: SW7..SW0 is used as threshold value, SW17 as on/off to save the file with the current image to a text file on the host system, and SW16-SW14 are used to specify the median filter length. The minimum length is 3. The edge detection run in a forever loop so that you can try different thresholds and filter lengths without the need to transfer the image again. Since the image transfer goes over the JTAG cable the time taken is substantial.

If you want to modify the project here is a brief description of the major files of the projects:

- If you like to test other images you need to convert your 320×240 BMP image using the MatLab script `bm2txt.m` or the factory provided `PrintNum.exe` to an MIF file. You find these files in the `c-source` subdi-

rectory. You do not need to recompile the Quartus design if you just change the image; you can use the SOF file provided.

- `median.c` is the program that includes all median filtering, adding of S&P noise, and file I/O. SW and LEDs are used too. Make sure the correct base address is used if you make changes to the hardware files. You do not need to recompile the Quartus design if you only change the software and you can use the SOF file provided.
- `DE2_115_ImageProcessing.qpf` is the top level project file that includes the `Qsys` Nios system as well as all connections to the I/O pins. Making modification to the `Qsys` design should only be done by an experienced `Qsys` user. As a minimum you should have completed the `Qsys` tutorial: “Introduction to the Altera `Qsys` System Integration Tool” from the University Program tutorials. You also need to download and install the University program IP cores that come with the free “University Program Installer.” The IP version must match the Quartus version you are using. After successful installation the IP blocks should appear in the `Qsys` component library as shown in Fig. 10.19, p. 778 on the left. Only after successful installation of the IP blocks will you be able to make modifications to the `Qsys` file. If you want to use another board make sure you include the correct pin file and make the required correction to the top level VHDL file `DE2_115_ImageProcessing.vhd`.

After you make any changes to the `Qsys` project make sure to generate the new Nios II system and recompile the whole project before downloading the SOF file to the board.

Custom Instruction Computer Project Files

The custom instruction computer project supports the CI for the bit swap operation found in Chap. 9 and the CI to improve the motion estimation in Chap. 10. The Quartus design is based on the Basic Computer system provided by Altera’s University Program. To get started with the CI project connect the DE2 to your PC via USB cable and add a power supply for the board. No VGA cable or VGA monitor is required for this project. Now copy the whole directory `DE2_115_CI_Computer` from the CD to your PC, memory stick, or network drive, and you are ready to start the Quartus software and load the project or double click `DE2_115_CI_Computer.qpf`. You should then download the project `DE2_115_CI_Computer.sof` to the board. You may try to import the whole software project from the `software/Motion` folder; however we found that most often pathes or files could not be found in the right location. It is usually more successful if you start with a “Hello World” project and then copy the required files in this hello world project. Therefore, we recommend that you within Eclipse generate a “Hello World” source code project, but name it project `Motion`. Right click the project and select `Run`

As → Nios II Hardware and it will produce in the terminal window the message “Hello from Nios II!” After you have successfully run this program replace the `hello_world.c` file with the project file you want to run. You can find these files in the `c-source` subdirectory. If you want to modify the project here is a brief description of the major files of the project:

- The software programs in the `c-source` subdirectory are: the `my_swap.c` file used in Chap. 9 for the three versions of the bit swap operation; the `madtest.c` file that has a brief check of byte access used for the MAD computation; the `motion.c` program that generates two test images and computes the motion vectors and measures the run time. You do not need to recompile the Quartus II or generate a `Qsys` system if you change only the software and you can use the provided SOF file.
- `DE2_115_CI_Processor.qpf` is the top level project file that includes the `Qsys` Nios system as well as all connections to the I/O pins. Making modification to the `Qsys` design should only be done by an experienced `Qsys` user. As a minimum you should have completed the `Qsys` tutorial: “Introduction to the Altera `Qsys` System Integration Tool” and “Making `Qsys` Components” from the University Program tutorials. You also need to download and install the University program IP cores that come with the free “University Program Installer.” The IP version must match the Quartus version you are using. After successful installation the IP blocks should appear in the `Qsys` component library as shown in Fig. 10.19, p. 778 on the left. Only after successful installation of the IP blocks will you be able to make modifications to the `Qsys` file. If you want to use another board make sure you include the correct pin file and make the required correction to the top level VHDL file `DE2_115_ImageProcessing.vhd`. The TCL scripts and the VHDL source code for the CI files can be found under `nios_system` →`synthesis`→`submodules`.

After you make any changes to the `Qsys` project make sure to generate the new Nios II system and recompile the whole project before downloading the SOF file to the board.

Appendix E. Glossary

ACC	Accumulator
ACT	Actel FPGA family
ADC	Analog-to-digital converter
ADCL	All-digital CL
ADF	Adaptive digital filter
ADPCM	Adaptive differential pulse code modulation
ADPLL	All-digital PLL
ADSP	Analog Devices digital signal processor family
AES	Advanced encryption standard
AFT	Arithmetic Fourier transform
AHDL	Altera HDL
AHSM	Additive half square multiplier
ALM	Adaptive logic module
ALU	Arithmetic logic unit
AM	Amplitude modulation
AMBA	Advanced microprocessor bus architecture
AMD	Advanced Micro Devices, Inc.
AMUSE	Algorithm for multiple unknown signals extraction
APEX	Adaptive principal component extraction
ASCII	American Standard Code for Information Interchange
ASIC	Application specific IC
AWGN	Additive white Gaussian noise
BCD	Binary coded decimal
BDD	Binary decision diagram
BIT	Binary digit
BLMS	Block LMS
BMP	Bitmap
BP	Bandpass
BRAM	Block RAM
BRS	Base removal scaling
BS	Barrelshifter
BSS	Blind source separation

CAD	Computer-aided design
CAE	Computer-aided engineering
CAM	Content addressable memory
CAST	Carlisle Adams and Stafford Tavares
CBC	Cipher block chaining
CBIC	Cell-based IC
CCD	Charge-coupled device
CCITT	Comité consultatif international téléphonique et télégraphique
CD	Compact disc
CFA	Common factor algorithm
CFB	Cipher feedback
CHF	Swiss franc
CIC	Cascaded integrator comb
CIF	Common intermediate format
CISC	Complex instruction set computer
CL	Costas loop
CLB	Configurable logic block
C-MOMS	Causal MOMS
CMOS	Complementary metal oxide semiconductor
CODEC	Coder/decoder
CORDIC	Coordinate rotation digital computer
COTS	Commercial off-the-shelf technology
CPLD	Complex PLD
CPU	Central processing unit
CQF	Conjugate quadrature filter
CRNS	Complex RNS
CRT	Chinese remainder theorem
CRT	Cathode ray tube
CSE	Common sub-expression
CSOC	Canonical self-orthogonal code
CSD	Canonical signed digit
CWT	Continuous wavelet transform
CZT	Chirp- z transform
DA	Distributed arithmetic
DAC	Digital-to-analog converter
DAT	Digital audio tap
DB	Daubechies filter
DC	Direct current
DCO	Digital controlled oscillator
DCT	Discrete cosine transform
DCU	Data cache unit
DDRAM	Double data rate RAM
DES	Data encryption standard
DFT	Discrete Fourier transform

DHT	Discrete Hartley transform
DIF	Decimation in frequency
DIT	Decimation in time
DLMS	Delayed LMS
DM	Delta modulation
DMA	Direct memory access
DMIPS	Dhrystone MIPS
DMT	Discrete Morlet transform
DOD	Department of defence
DPCM	Differential PCM
DPLL	Digital PLL
DSP	Digital signal processing
DST	Discrete sine transform
DWT	Discrete wavelet transform
EAB	Embedded array block
EASI	Equivariant adaptive separation via independence
ECB	Electronic code book
ECG	Electrocardiography
ECL	Emitter coupled logic
EDA	Electronic design automation
EDIF	Electronic design interchange format
EFF	Electronic Frontier Foundation
EOB	End of block
EPF	Altera FPGA family
EPROM	Electrically programmable ROM
ERA	Plessey FPGA family
ERNS	Eisenstein RNS
ESA	European Space Agency
ESB	Embedded system block
EVR	Eigenvalue ratio
EXU	Execution unit
FAEST	Fast a posteriori error sequential technique
FCT	Fast Cosine transform
FC2	FPGA compiler II
FF	Flip-flop
FFT	Fast Fourier transform
FIFO	First-in first-out
FIR	Finite impulse response
FIT	Fused internal timer
FLEX	Altera FPGA family
FM	Frequency modulation
FNT	Fermat NTT
FPGA	Field-programmable gate array

FPL	Field-programmable logic (combines CPLD and FPGA)
FPLD	FPL device
FPMAC	Floating-point MAC
FPS	Frames per second
FSF	Frequency sampling filter
FSK	Frequency shift keying
FSM	Finite state machine
GAL	Generic array logic
GF	Galois field
GFPMACS	Giga FPMAC
GIF	Graphic interchange format
GNU	GNU's not Unix
hpp	General purpose processor
GPR	General purpose register
HB	Half-band filter
HDL	Hardware description language
HDMI	High definition multimedia interface
HDTV	High-definition television
HI	High frequency
HP	Hewlett Packard
HSP	Harris Semiconductor DSP ICs
HW	Hardware
IBM	International Business Machines (corporation)
IC	Integrated circuit
ICA	Independent component analysis
ICU	Instruction cache unit
IDCT	Inverse DCT
IDEA	International data encryption algorithm
IDFT	Inverse discrete Fourier transform
IEC	International electrotechnical commission
IEEE	Institute of Electrical and Electronics Engineers
IF	Inter frequency
IFFT	Inverse fast Fourier transform
IIR	Infinite impulse response
IMA	Interactive multimedia association
I-MOMS	Interpolating MOMS
INTT	Inverse NTT
IP	Intellectual property
I/Q	In-/Quadrature phase
ISA	Instruction set architecture
ISDN	Integrated services digital network
ISO	International standardization organization

ITU	International Telecommunication Union
JPEG	Joint photographic experts group
JTAG	Joint test action group
KCPSM	Ken Chapman PSM
KLT	Karhunen–Loeve transform
LAB	Logic array block
LAN	Local area network
LC	Logic cell
LCD	Liquid-crystal display
LE	Logic element
LIFO	Last in first out
LISA	Language for instruction set architecture
LF	Low frequency
LFSR	Linear feedback shift register
LMS	Least-mean-square
LNS	Logarithmic number system
LO	Low frequency
LP	Low pass
LPC	Linear predictive coding
LPM	Library of parameterized modules
LRS	serial left right shifter
LS	Least-square
LSB	Least significant bit
LSI	Large scale integration
LTI	Linear time-invariant
LUT	Look-up table
LWDF	Lattice WDF
LZW	Lempel-Ziv-Welch
MAC	Multiplication and accumulate
MACH	AMD/Vantis FPGA family
MAG	Multiplier adder graph
MAX	Altera CPLD family
MIF	Memory initialization file
MIPS	Microprocessor without interlocked pipeline
MIPS	Million instructions per second
MLSE	Maximum likelihood sequence estimator
MMU	Memory management unit
MMX	Multimedia extension
MNT	Mersenne NTT
MOMS	Maximum order minimum support
μ P	Microprocessor

MPEG	Moving picture experts group
MPX	Multiplexer
MSPS	Millions of sample per second
MRC	Mixed radix conversion
MSB	Most significant bit
MUL	Multiplication
NCO	Numeric controlled oscillators
NLMS	Normalized LMS
NOF	Non-output fundamental
NP	Nonpolynomial complex problem
NRE	Nonrecurring engineering costs
NTSC	National television system committee
NTT	Number theoretic transform
OFB	Open feedback (mode)
O-MOMS	Optimal MOMS
OPAST	Orthogonal PAST
PAL	Phase alternating line
PAM	Pulse-amplitude-modulated
PAST	Projection approximation subspace tracking
PC	Personal computer
PC	Principle component
PCA	Principle component analysis
PCI	Peripheral component interconnect
PCM	Pulse-code modulation
PD	Phase detector
PDF	Probability density function
PDSP	Programmable digital signal processor
PFA	Prime factor algorithm
PIT	Programmable interval timer
PLA	Programmable logic array
PLD	Programmable logic device
PLL	Phase-locked loop
PM	Phase modulation
PNG	Portable network graphic
PPC	Power PC
PREP	Programmable Electronic Performance (cooperation)
PRNS	Polynomial RNS
PROM	Programmable ROM
PSK	Phase shift keying
PSM	Programmable state machine
QCIF	Quarter CIF

QDFT	Quantized DFT
QLI	Quick look-in
QFFT	Quantized FFT
QMFI	Quadrature mirror filter
QRNS	Quadratic RNS
QSM	Quarter square multiplier
QVGA	Quarter VGA
RAG	Reduced adder graph
RAM	Random access memory
RC	Resistor/capacity
RF	Radio frequency
RGB	Red, green and blue
RISC	Reduced instruction set computer
RLS	Recursive least square
RNS	Residue number system
ROM	Read only memory
RPFA	Rader prime factor algorithm
RS	serial right shifter
RSA	Rivest, Shamir, and Adelman
SD	Signed digit
SDRAM	Synchronous dynamic RAM
SECAM	Sequential color with memory
SG	Stochastic gradient
SIMD	Single instruction multiple data
SLMS	Signed LMS
SM	Signed magnitude
SNR	Signal-to-noise ratio
SOBI	Second order blind identification
SPEC	System performance evaluation cooperation
SPLD	Simple PLD
SPT	Signed power of two
SR	Shift register
SRAM	Static random access memory
SSE	Streaming SIMD extension
STFT	Short term Fourier transform
SVGA	Super VGA
SW	Software
SXGA	Super extended graphics array
TDLMS	Transform domain LMS
TLB	Translation look-aside buffer
TLU	Table look-up
TMS	Texas Instruments DSP family

TI	Texas Instruments
TOS	Top of stack
TSMC	Taiwan semiconductor manufacturing company
TTL	Transistor transistor logic
TVP	True vector processor
UART	Universal asynchronous receiver/transmitter
USB	Universal serial bus
VCO	Voltage-control oscillator
VGA	Video graphics array
VHDL	VHSIC hardware description language
VHSIC	Very high speed integrated circuit
VLC	Variable run-length coding
VLIW	Very long instruction word
VLSI	Very large integrated ICs
WDF	Wave digital filter
WDT	Watchdog timer
WFTA	Winograd Fourier transform algorithm
WSS	Wide sense stationary
WWW	World wide web
XC	Xilinx FPGA family
XNOR	exclusive NOR gate
YACC	Yet another compiler-compiler

References

1. B. Dipert: “EDN’s first annual PLD directory,” *EDN* pp. 54–84 (2000)
2. S. Brown, Z. Vranesic: *Fundamentals of Digital Logic with VHDL Design* (McGraw-Hill, New York, 1999)
3. D. Smith: *HDL Chip Design* (Doone Publications, Madison, Alabama, USA, 1996)
4. U. Meyer-Bäse: *The Use of Complex Algorithm in the Realization of Universal Sampling Receiver using FPGAs (in German)* (VDI/Springer, Düsseldorf, 1995), vol. 10, No. 404, 215 pages
5. U. Meyer-Bäse: *Fast Digital Signal Processing (in German)* (Springer, Heidelberg, 1999), 370 pages
6. P. Lapsley, J. Bier, A. Shoham, E. Lee: *DSP Processor Fundamentals* (IEEE Press, New York, 1997)
7. D. Shear: “EDN’s DSP Benchmarks,” *EDN* **33**, pp. 126–148 (1988)
8. V. Betz, S. Brown: “FPGA Challenges and Opportunities at 40 nm and Beyond,” in *International Conference on Field Programmable Logic and ApplicationsPrague* (2009), p. 4, fPL
9. Plessey: (1990), “Data sheet,” ERA60100
10. J. Greene, E. Hamdy, S. Beal: “Antifuse Field Programmable Gate Arrays,” *Proceedings of the IEEE* pp. 1042–56 (1993)
11. Lattice: (1997), “Data sheet,” GAL 16V8
12. J. Rose, A. Gamal, A. Sangiovanni-Vincentelli: “Architecture of Field-Programmable Gate Arrays,” *Proceedings of the IEEE* pp. 1013–29 (1993)
13. Xilinx: “PREP Benchmark Observations,” in *Xilinx-SeminarSan Jose* (1993)
14. Altera: “PREP Benchmarks Reveal FLEX 8000 is Biggest, MAX 7000 is Fastest,” in *Altera News & Views San Jose* (1993)
15. Actel: “PREP Benchmarks Confirm Cost Effectiveness of Field Programmable Gate Arrays,” in *Actel-Seminar* (1993)
16. E. Lee: “Programmable DSP Architectures: Part I,” *IEEE Transactions on Acoustics, Speech and Signal Processing Magazine* pp. 4–19 (1988)
17. E. Lee: “Programmable DSP Architectures: Part II,” *IEEE Transactions on Acoustics, Speech and Signal Processing Magazine* pp. 4–14 (1989)
18. R. Petersen, B. Hutchings: “An Assessment of the Suitability of FPGA-Based Systems for Use in Digital Signal Processing,” *Lecture Notes in Computer Science* **975**, 293–302 (1995)
19. J. Villasenor, B. Hutchings: “The Flexibility of Configurable Computing,” *IEEE Signal Processing Magazine* pp. 67–84 (1998)
20. Altera: (2011), “Floating-Point Megafunctions User Guide,” ver. 11.1
21. Texas Instruments: (2008), “TMS320C6727B, TMS320C6726B, TMS320C6722B, TMS320C6720 Floating-Point Digital Signal Processors”
22. Xilinx: (1993), “Data book,” XC2000, XC3000 and XC4000
23. Altera: (1996), “Data sheet,” FLEX 10K CPLD Family

24. Altera: (2013), “Cyclone IV Device Handbook,” volume 1-3
25. U. Meyer-Bäse: *Digital Signal Processing with Field Programmable Gate Arrays*, 1st edn. (Springer, Heidelberg, 2001), 422 pages
26. F. Vahid, T. Givargis: *Embedded System Design* (John Wiley & Sons, New York, 2001)
27. J. Hakewill: “Gainin Control over Silicon IP,” *Communication Design* online (2000)
28. E. Castillo, U. Meyer-Baese, L. Parrilla, A. Garcia, A. Lloris: “Watermarking Strategies for RNS-Based System Intellectual Property Protection,” in *Proc. of 2005 IEEE Workshop on Signal Processing Systems SiPS’05* Athens (2005), pp. 160–165
29. O. Spaniol: *Computer Arithmetic: Logic and Design* (John Wiley & Sons, New York, 1981)
30. I. Koren: *Computer Arithmetic Algorithms* (Prentice Hall, Englewood Cliffs, New Jersey, 1993)
31. E.E. Swartzlander: *Computer Arithmetic, Vol. I* (Dowden, Hutchinson and Ross, Inc., Stroudsburg, Pennsylvania, 1980), also reprinted by IEEE Computer Society Press 1990
32. E. Swartzlander: *Computer Arithmetic, Vol. II* (IEEE Computer Society Press, Stroudsburg, Pennsylvania, 1990)
33. K. Hwang: *Computer Arithmetic: Principles, Architecture and Design* (John Wiley & Sons, New York, 1979)
34. U. Meyer-Baese: *Digital Signal Processing with Field Programmable Gate Arrays*, 3rd edn. (Springer-Verlag, Berlin, 2007), 774 pages
35. N. Takagi, H. Yasuura, S. Yajima: “High Speed VLSI multiplication algorithm with a redundant binary addition tree,” *IEEE Transactions on Computers* **34**(2) (1985)
36. D. Bull, D. Horrocks: “Reduced-Complexity Digital Filtering Structures using Primitive Operations,” *Electronics Letters* pp. 769–771 (1987)
37. D. Bull, D. Horrocks: “Primitive operator digital filters,” *IEE Proceedings-G* **138**, 401–411 (1991)
38. A. Dempster, M. Macleod: “Use of Minimum-Adder Multiplier Blocks in FIR Digital Filters,” *IEEE Transactions on Circuits and Systems II* **42**, 569–577 (1995)
39. A. Dempster, M. Macleod: “Comments on “Minimum Number of Adders for Implementing a Multiplier and Its Application to the Design of Multiplierless Digital Filters”,” *IEEE Transactions on Circuits and Systems II* **45**, 242–243 (1998)
40. F. Taylor, R. Gill, J. Joseph, J. Radke: “A 20 Bit Logarithmic Number System Processor,” *IEEE Transactions on Computers* **37**(2) (1988)
41. P. Lee: “An FPGA Prototype for a Multiplierless FIR Filter Built Using the Logarithmic Number System,” *Lecture Notes in Computer Science* **975**, 303–310 (1995)
42. J. Mitchell: “Computer multiplication and division using binary logarithms,” *IRE Transactions on Electronic Computers* **EC-11**, 512–517 (1962)
43. N. Szabo, R. Tanaka: *Residue Arithmetic and its Applications to Computer Technology* (McGraw-Hill, New York, 1967)
44. M. Soderstrand, W. Jenkins, G. Jullien, F. Taylor: *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing*, IEEE Press Reprint Series (IEEE Press, New York, 1986)
45. U. Meyer-Bäse, A. Meyer-Bäse, J. Mellott, F. Taylor: “A Fast Modified CORDIC-Implementation of Radial Basis Neural Networks,” *Journal of VLSI Signal Processing* pp. 211–218 (1998)

46. V. Hamann, M. Sprachmann: "Fast Residual Arithmetics with FPGAs," in *Proceedings of the Workshop on Design Methodologies for Microelectronics* Smolenice Castle, Slovakia (1995), pp. 253–255
47. G. Jullien: "Residue Number Scaling and Other Operations Using ROM Arrays," *IEEE Transactions on Communications* **27**, 325–336 (1978)
48. M. Griffin, M. Sousa, F. Taylor: "Efficient Scaling in the Residue Number System," in *IEEE International Conference on Acoustics, Speech, and Signal Processing* (1989), pp. 1075–1078
49. G. Zelniker, F. Taylor: "A Reduced-Complexity Finite Field ALU," *IEEE Transactions on Circuits and Systems* **38**(12), 1571–1573 (1991)
50. IEEE: "Standard for Binary Floating-Point Arithmetic," *IEEE Std 754-1985* pp. 1–14 (1985)
51. IEEE: "Standard for Binary Floating-Point Arithmetic," *IEEE Std 754-2008* pp. 1–70 (2008)
52. N. Shirazi, P. Athanas, A. Abbott: "Implementation of a 2-D Fast Fourier Transform on an FPGA-Based Custom Computing Machine," *Lecture Notes in Computer Science* **975**, 282–292 (1995)
53. Xilinx: "Using the Dedicated Carry Logic in XC4000E," in *Xilinx Application Note XAPP 013* San Jose (1996)
54. M. Bayoumi, G. Jullien, W. Miller: "A VLSI Implementation of Residue Adders," *IEEE Transactions on Circuits and Systems* pp. 284–288 (1987)
55. A. Garcia, U. Meyer-Bäse, F. Taylor: "Pipelined Hogenauer CIC Filters using Field-Programmable Logic and Residue Number System," in *IEEE International Conference on Acoustics, Speech, and Signal Processing* Vol. 5 (1998), pp. 3085–3088
56. L. Turner, P. Graumann, S. Gibb: "Bit-serial FIR Filters with CSD Coefficients for FPGAs," *Lecture Notes in Computer Science* **975**, 311–320 (1995)
57. J. Logan: "A Square-Summing, High-Speed Multiplier," *Computer Design* pp. 67–70 (1971)
58. Leibowitz: "A Simplified Binary Arithmetic for the Fermat Number Transform," *IEEE Transactions on Acoustics, Speech and Signal Processing* **24**, 356–359 (1976)
59. T. Chen: "A Binary Multiplication Scheme Based on Squaring," *IEEE Transactions on Computers* pp. 678–680 (1971)
60. E. Johnson: "A Digital Quarter Square Multiplier," *IEEE Transactions on Computers* pp. 258–260 (1980)
61. Altera: (2004), "Implementing Multipliers in FPGA Devices," application note 306, Ver. 3.0
62. D. Anderson, J. Earle, R. Goldschmidt, D. Powers: "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM Journal of Research and Development* **11**, 34–53 (1967)
63. V. Pedroni: *Circuit Design and Simulation with VHDL* (The MIT Press, Cambridge, Massachusetts, 2010)
64. A. Rushton: *VHDL for logic Synthesis*, 3rd edn. (John Wiley & Sons, New York, 2011)
65. P. Ashenden: *The Designer's Guide to VHDL*, 3rd edn. (Morgan Kaufman Publishers, Inc., San Mateo, CA, 2008)
66. A. Croisier, D. Esteban, M. Levilion, V. Rizo: (1973), "Digital Filter for PCM Encoded Signals," US patent no. 3777130
67. A. Peled, B. Liu: "A New Realization of Digital Filters," *IEEE Transactions on Acoustics, Speech and Signal Processing* **22**(6), 456–462 (1974)
68. K. Yiu: "On Sign-Bit Assignment for a Vector Multiplier," *Proceedings of the IEEE* **64**, 372–373 (1976)

69. K. Kammeyer: "Quantization Error on the Distributed Arithmetic," *IEEE Transactions on Circuits and Systems* **24**(12), 681–689 (1981)
70. F. Taylor: "An Analysis of the Distributed-Arithmetic Digital Filter," *IEEE Transactions on Acoustics, Speech and Signal Processing* **35**(5), 1165–1170 (1986)
71. S. White: "Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review," *IEEE Transactions on Acoustics, Speech and Signal Processing Magazine* pp. 4–19 (1989)
72. K. Kammeyer: "Digital Filter Realization in Distributed Arithmetic," in *Proc. European Conf. on Circuit Theory and Design* (1976), Genoa, Italy
73. F. Taylor: *Digital Filter Design Handbook* (Marcel Dekker, New York, 1983)
74. H. Nussbaumer: *Fast Fourier Transform and Convolution Algorithms* (Springer, Heidelberg, 1990)
75. H. Schmid: *Decimal Computation* (John Wiley & Sons, New York, 1974)
76. Y. Hu: "CORDIC-Based VLSI Architectures for Digital Signal Processing," *IEEE Signal Processing Magazine* pp. 16–35 (1992)
77. U. Meyer-Bäse, A. Meyer-Bäse, W. Hilberg: "COordinate Rotation DIgital Computer (CORDIC) Synthesis for FPGA," *Lecture Notes in Computer Science* **849**, 397–408 (1994)
78. J.E. Volder: "The CORDIC Trigonometric computing technique," *IRE Transactions on Electronics Computers* **8**(3), 330–4 (1959)
79. J. Walther: "A Unified algorithm for elementary functions," *Spring Joint Computer Conference* pp. 379–385 (1971)
80. X. Hu, R. Huber, S. Bass: "Expanding the Range of Convergence of the CORDIC Algorithm," *IEEE Transactions on Computers* **40**(1), 13–21 (1991)
81. D. Timmermann (1990): "CORDIC-Algorithmen, Architekturen und monolithische Realisierungen mit Anwendungen in der Bildverarbeitung," Ph.D. thesis, VDI Press, Serie 10, No. 152
82. H. Hahn (1991): "Untersuchung und Integration von Berechnungsverfahren elementarer Funktionen auf CORDIC-Basis mit Anwendungen in der adaptiven Signalverarbeitung," Ph.D. thesis, VDI Press, Serie 9, No. 125
83. G. Ma (1989): "A Systolic Distributed Arithmetic Computing Machine for Digital Signal Processing and Linear Algebra Applications," Ph.D. thesis, University of Florida, Gainesville
84. Y.H. Hu: "The Quantization Effects of the CORDIC-Algorithm," *IEEE Transactions on signal processing* pp. 834–844 (1992)
85. M. Abramowitz, A. Stegun: *Handbook of Mathematical Functions*, 9th edn. (Dover Publications, Inc., New York, 1970)
86. W. Press, W. Teukolsky, W. Vetterling, B. Flannery: *Numerical Recipes in C*, 2nd edn. (Cambridge University Press, Cambridge, 1992)
87. Intersil: (2001), "Data sheet," HSP50110
88. A.V. Oppenheim, R.W. Schafer: *Discrete-Time Signal Processing* (Prentice Hall, Englewood Cliffs, New Jersey, 1992)
89. D.J. Goodman, M.J. Carey: "Nine Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech and Signal Processing* pp. 121–126 (1977)
90. U. Meyer-Baese, J. Chen, C. Chang, A. Dempster: "A Comparison of Pipelined RAG-n and DA FPGA-Based Multiplierless Filters," in *IEEE Asia Pacific Conference on Circuits and Systems* (2006), pp. 1555–1558
91. O. Gustafsson, A. Dempster, L. Wanhammar: "Extended Results for Minimum-Adder Constant Integer Multipliers," in *IEEE International Conference on Acoustics, Speech, and Signal ProcessingPhoenix* (2002), pp. 73–76

92. Y. Wang, K. Roy: "CSDC: A New Complexity Reduction Technique for Multiplierless Implementation of Digital FIR Filters," *IEEE Transactions on Circuits and Systems I* **52**(0), 1845–1852 (2005)
93. H. Samueli: "An Improved Search Algorithm for the Design of Multiplierless FIR Filters with Powers-of-Two Coefficients," *IEEE Transactions on Circuits and Systems* **36**(7), 10441047 (1989)
94. Y. Lim, S. Parker: "Discrete Coefficient FIR Digital Filter Design Based Upon an LMS Criteria," *IEEE Transactions on Circuits and Systems* **36**(10), 723–739 (1983)
95. Altera: (2013), "FIR Compiler: MegaCore Function User Guide," ver. 12.1
96. U. Meyer-Baese, G. Botella, D. Romero, M. Kumm: "Optimization of high speed pipelining in FPGA-based FIR filter design using genetic algorithm," in *Proc. SPIE Int. Soc. Opt. Eng., Independent Component Analyses, Wavelets, Neural Networks, Biosystems, and Nanoengineering X* (2012), pp. 84010R1–12, vol. 8401
97. R. Hartley: "Subexpression Sharing in Filters Using Canonic Signed Digital Multiplier," *IEEE Transactions on Circuits and Systems II* **30**(10), 677–688 (1996)
98. S. Mirzaei, R. Kastner, A. Hosangadi: "Layout aware optimization of high speed fixed coefficient FIR filters for FPGAs," *International Journal of Reconfigurable Computing* **2010**(3), 1–17 (2010)
99. R. Saal: *Handbook of filter design* (AEG-Telefunken, Frankfurt, Germany, 1979)
100. C. Barnes, A. Fam: "Minimum Norm Recursive Digital Filters that Are Free of Overflow Limit Cycles," *IEEE Transactions on Circuits and Systems* pp. 569–574 (1977)
101. A. Fettweis: "Wave Digital Filters: Theorie and Practice," *Proceedings of the IEEE* pp. 270–327 (1986)
102. R. Crochiere, A. Oppenheim: "Analysis of Linear Digital Networks," *Proceedings of the IEEE* **63**(4), 581–595 (1975)
103. A. Dempster, M. Macleod: "Multiplier blocks and complexity of IIR structures," *Electronics Letters* **30**(22), 1841–1842 (1994)
104. A. Dempster, M. Macleod: "IIR Digital Filter Design Using Minimum Adder Multiplier Blocks," *IEEE Transactions on Circuits and Systems II* **45**, 761–763 (1998)
105. A. Dempster, M. Macleod: "Constant Integer Multiplication using Minimum Adders," *IEE Proceedings - Circuits, Devices & Systems* **141**, 407–413 (1994)
106. K. Parhi, D. Messerschmidt: "Pipeline Interleaving and Parallelism in Recursive Digital Filters - Part I: Pipelining Using Scattered Look-Ahead and Decomposition," *IEEE Transactions on Acoustics, Speech and Signal Processing* **37**(7), 1099–1117 (1989)
107. H. Loomis, B. Sinha: "High Speed Recursive Digital Filter Realization," *Circuits, Systems, Signal Processing* **3**(3), 267–294 (1984)
108. M. Soderstrand, A. de la Serna, H. Loomis: "New Approach to Clustered Look-ahead Pipelined IIR Digital Filters," *IEEE Transactions on Circuits and Systems II* **42**(4), 269–274 (1995)
109. J. Living, B. Al-Hashimi: "Mixed Arithmetic Architecture: A Solution to the Iteration Bound for Resource Efficient FPGA and CPLD Recursive Digital Filters," in *IEEE International Symposium on Circuits and Systems* Vol. I (1999), pp. 478–481
110. H. Martinez, T. Parks: "A Class of Infinite-Duration Impulse Response Digital Filters for Sampling Rate Reduction," *IEEE Transactions on Acoustics, Speech and Signal Processing* **26**(4), 154–162 (1979)

111. K. Parhi, D. Messerschmidt: "Pipeline Interleaving and Parallelism in Recursive Digital Filters - Part II: Pipelined Incremental Block Filtering," *IEEE Transactions on Acoustics, Speech and Signal Processing* **37**(7), 1118–1134 (1989)
112. A. Gray, J. Markel: "Digital Lattice and Ladder Filter Synthesis," *IEEE Transactions on Audio and Electroacoustics* **21**(6), 491–500 (1973)
113. L. Gazsi: "Explicit Formulas for Lattice Wave Digital Filters," *IEEE Transactions on Circuits and Systems* pp. 68–88 (1985)
114. J. Xu, U. Meyer-Baese, K. Huang: "FPGA-based solution for real-time tracking of time-varying harmonics and power disturbances," *International Journal of Power Electronics (IJPELEC)* **4**(2), 134–159 (2012)
115. J. Xu (2009): "FPGA-based Real Time Processing of Time-Varying Waveform Distortions and Power Disturbances in Power Systems," Ph.D. thesis, Florida State University
116. L. Jackson: "Roundoff-Noise Analysis for Fixed-point Digital Filters Realized in Cascade or Parallel Form," *IEEE Transactions on Audio and Electroacoustics* **18**(2), 107–123 (1970)
117. W. Hess: *Digitale Filter* (Teubner Studienbücher, Stuttgart, 1989)
118. H.L. Arriens: (2013), "(L)WDF Toolbox for MATLAB," personal communication
URL <http://ens.ewi.tudelft.nl/~huib/mtoolbox/>
119. T. Saramaki: "On the Design of Digital Filters as a Sum of Two All-pass Filters," *IEEE Transactions on Circuits and Systems* pp. 1191–1193 (1985)
120. P. Vaidyanathan, P. Regalia, S. Mitra: "Design of doubly complementary IIR digital filters using a single complex allpass filter, with multirate applications," *IEEE Transactions on Circuits and Systems* **34**, 378–389 (1987)
121. M. Anderson, S. Summerfield, S. Lawson: "Realisation of lattice wave digital filters using three-port adaptors," *IEE Electronics Letters* pp. 628–629 (1995)
122. M. Shajaan, J. Sorensen: "Time-Area Efficient Multiplier-Free Recursive Filter Architectures for FPGA Implementation," in *IEEE International Conference on Acoustics, Speech, and Signal Processing* (1996), pp. 3269–3272
123. P. Vaidyanathan: *Multirate Systems and Filter Banks* (Prentice Hall, Englewood Cliffs, New Jersey, 1993)
124. S. Winograd: "On Computing the Discrete Fourier Transform," *Mathematics of Computation* **32**, 175–199 (1978)
125. Z. Mou, P. Duhamel: "Short-Length FIR Filters and Their Use in Fast Non-recursive Filtering," *IEEE Transactions on Signal Processing* **39**, 1322–1332 (1991)
126. P. Balla, A. Antoniou, S. Morgera: "Higher Radix Aperiodic-Convolution Algorithms," *IEEE Transactions on Acoustics, Speech and Signal Processing* **34**(1), 60–68 (1986)
127. E.B. Hogenauer: "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech and Signal Processing* **29**(2), 155–162 (1981)
128. Harris: (1992), "Data sheet," HSP43220 Decimating Digital Filter
129. Motorola: (1989), "Datasheet," DSP56ADC16 16-Bit Sigma-Delta Analog-to-Digital Converter
130. Intersil: (2000), "Data sheet," HSP50214 Programmable Downconverter
131. Texas Instruments: (2000), "Data sheet," GC4114 Quad Transmit Chip
132. Altera: (2007), "Understanding CIC Compensation Filters," application note 455, Ver. 1.0

133. O. Six (1996): "Design and Implementation of a Xilinx universal XC-4000 FPGAs board," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
134. S. Dworak (1996): "Design and Realization of a new Class of Frequency Sampling Filters for Speech Processing using FPGAs," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
135. L. Wang, W. Hsieh, T. Truong: "A Fast Computation of 2-D Cubic-spline Interpolation," *IEEE Signal Processing Letters* **11**(9), 768–771 (2004)
136. T. Laakso, V. Valimaki, M. Karjalainen, U. Laine: "Splitting the Unit Delay," *IEEE Signal Processing Magazine* **13**(1), 30–60 (1996)
137. M. Unser: "Splines: a Perfect Fit for Signal and Image Processing," *IEEE Signal Processing Magazine* **16**(6), 22–38 (1999)
138. S. Cucchi, F. Desinan, G. Parladori, G. Sicuranza: "DSP Implementation of Arbitrary Sampling Frequency Conversion for High Quality Sound Application," in *IEEE International Symposium on Circuits and Systems* Vol. 5 (1991), pp. 3609–3612
139. C. Farrow: "A Continuously Variable Digital Delay Element," in *IEEE International Symposium on Circuits and Systems* Vol. 3 (1988), pp. 2641–2645
140. S. Mitra: *Digital Signal Processing: A Computer-Based Approach*, 3rd edn. (McGraw Hill, Boston, 2006)
141. S. Dooley, R. Stewart, T. Durrani: "Fast On-line B-spline Interpolation," *IEE Electronics Letters* **35**(14), 1130–1131 (1999)
142. Altera: "Farrow-Based Decimating Sample Rate Converter," in *Altera application note AN-347* San Jose (2004)
143. F. Harris: "Performance and Design Considerations of the Farrow Filter when used for Arbitrary Resampling of Sampled Time Series," in *Conference Record of the Thirty-First Asilomar Conference on Signals, Systems & Computers* Vol. 2 (1997), pp. 1745–1749
144. M. Unser, A. Aldroubi, M. Eden: "B-spline Signal Processing: I– Theory," *IEEE Transactions on Signal Processing* **41**(2), 821–833 (1993)
145. P. Vaidyanathan: "Generalizations of the Sampling Theorem: Seven Decades after Nyquist," *Circuits and Systems I: Fundamental Theory and Applications* **48**(9), 1094–1109 (2001)
146. Z. Mihajlovic, A. Goluban, M. Zagar: "Frequency Domain Analysis of B-spline Interpolation," in *Proceedings of the IEEE International Symposium on Industrial Electronics* Vol. 1 (1999), pp. 193–198
147. M. Unser, A. Aldroubi, M. Eden: "Fast B-spline Transforms for Continuous Image Representation and Interpolation," *IEEE Transactions on Pattern Analysis and Machine Intelligence* **13**(3), 277–285 (1991)
148. M. Unser, A. Aldroubi, M. Eden: "B-spline Signal Processing: II– Efficiency Design and Applications," *IEEE Transactions on Signal Processing* **41**(2), 834–848 (1993)
149. M. Unser, M. Eden: "FIR Approximations of Inverse Filters and Perfect Reconstruction Filter Banks," *Signal Processing* **36**(2), 163–174 (1994)
150. T. Blu, P. Thévenaz, M. Unser: "MOMS: Maximal-Order Interpolation of Minimal Support," *IEEE Transactions on Image Processing* **10**(7), 1069–1080 (2001)
151. T. Blu, P. Thévenaz, M. Unser: "High-Quality Causal Interpolation for Online Unidimensional Signal Processing," in *Proceedings of the Twelfth European Signal Processing Conference (EUSIPCO'04)* (2004), pp. 1417–1420
152. A. Gotchev, J. Vesma, T. Saramäki, K. Egiazarian: "Modified B-Spline Functions for Efficient Image Interpolation," in *First IEEE Balkan Conference on*

- Signal Processing, Communications, Circuits, and Systems* (2000), pp. 241–244
153. W. Hawkins: “FFT Interpolation for Arbitrary Factors: a Comparison to Cubic Spline Interpolation and Linear Interpolation,” in *Proceedings IEEE Nuclear Science Symposium and Medical Imaging Conference* Vol. 3 (1994), pp. 1433–1437
 154. A. Haar: “Zur Theorie der orthogonalen Funktionensysteme,” *Mathematische Annalen* **69**, 331–371 (1910). Dissertation Göttingen 1909
 155. W. Sweldens: “The Lifting Scheme: A New Philosophy in Biorthogonal Wavelet Constructions,” in *SPIE, Wavelet Applications in Signal and Image Processing III* (1995), pp. 68–79
 156. C. Herley, M. Vetterli: “Wavelets and Recursive Filter Banks,” *IEEE Transactions on Signal Processing* **41**, 2536–2556 (1993)
 157. I. Daubechies: *Ten Lectures on Wavelets* (Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1992)
 158. I. Daubechies, W. Sweldens: “Factoring Wavelet Transforms into Lifting Steps,” *The Journal of Fourier Analysis and Applications* **4**, 365–374 (1998)
 159. G. Strang, T. Nguyen: *Wavelets and Filter Banks* (Wellesley-Cambridge Press, Wellesley MA, 1996)
 160. D. Esteban, C. Galand: “Applications of Quadrature Mirror Filters to Split Band Voice Coding Schemes,” in *IEEE International Conference on Acoustics, Speech, and Signal Processing* (1977), pp. 191–195
 161. M. Smith, T. Barnwell: “Exact Reconstruction Techniques for Tree-Structured Subband Coders,” *IEEE Transactions on Acoustics, Speech and Signal Processing* pp. 434–441 (1986)
 162. M. Vetterli, J. Kovacevic: *Wavelets and Subband Coding* (Prentice Hall, Englewood Cliffs, New Jersey, 1995)
 163. R. Crochiere, L. Rabiner: *Multirate Digital Signal Processing* (Prentice Hall, Englewood Cliffs, New Jersey, 1983)
 164. M. Achery, J.M. Mangen, Y. Buhler.: “Progressive Wavelet Algorithm versus JPEG for the Compression of METEOSAT Data,” in *SPIE, San Diego* (1995)
 165. T. Ebrahimi, M. Kunt: “Image Compression by Gabor Expansion,” *Optical Engineering* **30**, 873–880 (1991)
 166. D. Gabor: “Theory of communication,” *J. Inst. Elect. Eng (London)* **93**, 429–457 (1946)
 167. A. Grossmann, J. Morlet: “Decomposition of Hardy Functions into Square Integrable Wavelets of Constant Shape,” *SIAM J. Math. Anal.* **15**, 723–736 (1984)
 168. U. Meyer-Bäse: “High Speed Implementation of Gabor and Morlet Wavelet Filterbanks using RNS Frequency Sampling Filters,” in *Aerosense 98 *SPIE*, Orlando* (1998), pp. 522–533
 169. U. Meyer-Bäse: “Die Hutlets – eine biorthogonale Wavelet-Familie: Effiziente Realisierung durch multipliziererfreie, perfekt rekonstruierende Quadratur Mirror Filter,” *Frequenz* pp. 39–49 (1997)
 170. U. Meyer-Bäse, F. Taylor: “The Hutlets - a Biorthogonal Wavelet Family and their High Speed Implementation with RNS, Multiplier-free, Perfect Reconstruction QMF,” in *Aerosense 97 SPIE, Orlando* (1997), pp. 670–681
 171. D. Donoho, I. Johnstone: “Ideal Spatial Adatation by Wavelet Shrinkage,” *Biometrika* **81**(3), 425–545 (1994)
 172. S. Mallat: *A Wavelet Tour of Signal Processing* (Academic Press, San Diego, USA, 1998)
 173. D. Donoho, I. Johnstone, G. Kerkyacharian, D. Picard: “Wavelet Shrinkage: Asymptopia?”, *J. Roy. Statist. Soc.* **57**(2), 301–369 (1995)

174. M. Heideman, D. Johnson, C. Burrus: "Gauss and the History of the Fast Fourier Transform," *IEEE Transactions on Acoustics, Speech and Signal Processing Magazine* **34**, 265–267 (1985)
175. C. Burrus: "Index Mappings for Multidimensional Formulation of the DFT and Convolution," *IEEE Transactions on Acoustics, Speech and Signal Processing* **25**, 239–242 (1977)
176. B. Baas (1997): "An Approach to Low-power, High-performance, Fast Fourier Transform Processor Design," Ph.D. thesis, Stanford University
177. G. Sunada, J. Jin, M. Berzins, T. Chen: "COBRA: An 1.2 Million Transistor Exandable Column FFT Chip," in *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors* (IEEE Computer Society Press, Los Alamitos, CA, USA, 1994), pp. 546–550
178. TMS: (1996), "TM-66 swiFFT Chip," Texas Memory Systems
179. SHARP: (1997), "BDSP9124," digital signal processor
180. J. Mellott (1997): "Long Instruction Word Computer," Ph.D. thesis, University of Florida, Gainesville
181. P. Lavoie: "A High-Speed CMOS Implementation of the Winograd Fourier Transform Algorithm," *IEEE Transactions on Signal Processing* **44**(8), 2121–2126 (1996)
182. G. Panneerselvam, P. Graumann, L. Turner: "Implementation of Fast Fourier Transforms and Discrete Cosine Transforms in FPGAs," in *Lecture Notes in Computer Science* Vol. 1142 (1996), pp. 1142:272–281
183. Altera: "Fast Fourier Transform," in *Solution Brief 12, Altera Corporaration* (1997)
184. G. Goslin: "Using Xilinx FPGAs to Design Custom Digital Signal Processing Devices," in *Proceedings of the DSP^X* (1995), pp. 595–604
185. C. Dick: "Computing 2-D DFTs Using FPGAs," *Lecture Notes in Computer Science: Field-Programmable Logic* pp. 96–105 (1996)
186. S.D. Stearns, D.R. Hush: *Digital Signal Analysis* (Prentice Hall, Englewood Cliffs, New Jersey, 1990)
187. K. Kammeyer, K. Kroschel: *Digitale Signalverarbeitung* (Teubner Studienbücher, Stuttgart, 1989)
188. E. Brigham: *FFT*, 3rd edn. (Oldenbourg Verlag, München Wien, 1987)
189. R. Ramirez: *The FFT: Fundamentals and Concepts* (Prentice Hall, Englewood Cliffs, New Jersey, 1985)
190. R.E. Blahut: *Theory and practice of error control codes* (Addison-Wesley, Melo Park, California, 1984)
191. C. Burrus, T. Parks: *DFT/FFT and Convolution Algorithms* (John Wiley & Sons, New York, 1985)
192. D. Elliott, K. Rao: *Fast Transforms: Algorithms, Analyses, Applications* (Academic Press, New York, 1982)
193. A. Nuttall: "Some Windows with Very Good Sidelobe Behavior," *IEEE Transactions on Acoustics, Speech and Signal Processing* **ASSP-29**(1), 84–91 (1981)
194. U. Meyer-Bäse, K. Damm (1988): "Fast Fourier Transform using Signal Processor," Master's thesis, Department of Information Science, Darmstadt University of Technology
195. M. Narasimha, K. Shenoi, A. Peterson: "Quadratic Residues: Application to Chirp Filters and Discrete Fourier Transforms," in *IEEE International Conference on Acoustics, Speech, and Signal Processing* (1976), pp. 12–14
196. C. Rader: "Discrete Fourier Transform when the Number of Data Samples is Prime," *Proceedings of the IEEE* **56**, 1107–8 (1968)

197. J. McClellan, C. Rader: *Number Theory in Digital Signal Processing* (Prentice Hall, Englewood Cliffs, New Jersey, 1979)
198. I. Good: "The Relationship between Two Fast Fourier Transforms," *IEEE Transactions on Computers* **20**, 310–317 (1971)
199. L. Thomas: "Using a Computer to Solve Problems in Physics," in *Applications of Digital Computers* (Ginn, Dordrecht, 1963)
200. A. Dandalis, V. Prasanna: "Fast Parallel Implementation of DFT Using Configurable Devices," *Lecture Notes in Computer Science* **1304**, 314–323 (1997)
201. U. Meyer-Bäse, S. Wolf, J. Mellott, F. Taylor: "High Performance Implementation of Convolution on a Multi FPGA Board using NTT's defined over the Eisenstein Residuen Number System," in *Aerosense 97 SPIE, Orlando* (1997), pp. 431–442
202. Xilinx: (2000), "High-Performance 256-Point Complex FFT/IFFT," product specification
203. Altera: (2012), "FFT MegaCore Function: User Guide," UG-FFT-12.0
204. Z. Wang: "Fast Algorithms for the Discrete W transform and for the discrete Fourier Transform," *IEEE Transactions on Acoustics, Speech and Signal Processing* pp. 803–816 (1984)
205. M. Narasimha, A. Peterson: "On the Computation of the Discrete Cosine Transform," *IEEE Transaction on Communications* **26**(6), 934–936 (1978)
206. K. Rao, P. Yip: *Discrete Cosine Transform* (Academic Press, San Diego, CA, 1990)
207. B. Lee: "A New Algorithm to Compute the Discrete Cosine Transform," *IEEE Transactions on Acoustics, Speech and Signal Processing* **32**(6), 1243–1245 (1984)
208. S. Ramachandran, S. Srinivasan, R. Chen: "EPLD-Based Architecture of Real Time 2D-discrete Cosine Transform and Quantization for Image Compression," in *IEEE International Symposium on Circuits and Systems* Vol. III (1999), pp. 375–378
209. C. Burrus, P. Eschenbacher: "An In-Place, In-Order Prime Factor FFT Algorithm," *IEEE Transactions on Acoustics, Speech and Signal Processing* **29**(4), 806–817 (1981)
210. H. Lüke: *Signalübertragung* (Springer, Heidelberg, 1988)
211. D. Herold, R. Huthmann (1990): "Decoder for the Radio Data System (RDS) using Signal Processor TMS320C25," Master's thesis, Institute for Data Techniques, Darmstadt University of Technology
212. U. Meyer-Bäse, R. Watzel: "A comparison of DES and LFSR based FPGA Implementable Cryptography Algorithms," in *3rd International Symposium on Communication Theory & Applications* (1995), pp. 291–298
213. U. Meyer-Bäse, R. Watzel: "An Optimized Format for Long Frequency Paging Systems," in *3rd International Symposium on Communication Theory & Applications* (1995), pp. 78–79
214. U. Meyer-Bäse: "Convolutional Error Decoding with FPGAs," *Lecture Notes in Computer Science* **1142**, 376–175 (1996)
215. R. Watzel (1993): "Design of Paging Scheme and Implementation of the Suitable Crypto-Controller using FPGAs," Master's thesis, Institute for Data Techniques, Darmstadt University of Technology
216. J. Maier, T. Schubert (1993): "Design of Convolutional Decoders using FPGAs for Error Correction in a Paging System," Master's thesis, Institute for Data Techniques, Darmstadt University of Technology
217. Y. Gao, D. Herold, U. Meyer-Bäse: "Zum bestehenden Übertragungsprotokoll kompatible Fehlerkorrektur," in *Funkuhren Zeitsignale Normalfrequenzen* (1993), pp. 99–112

218. D. Herold (1991): "Investigation of Error Corrections Steps for DCF77 Signals using Programmable Gate Arrays," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
219. P. Sweeney: *Error Control Coding* (Prentice Hall, New York, 1991)
220. D. Wiggert: *Error-Control Coding and Applications* (Artech House, Dedham, Mass., 1988)
221. G. Clark, J. Cain: *Error-Correction Coding for Digital Communications* (Plenum Press, New York, 1988)
222. W. Stahnke: "Primitive Binary Polynomials," *Mathematics of Computation* pp. 977–980 (1973)
223. W. Fumy, H. Riess: *Kryptographie* (R. Oldenbourg Verlag, München, 1988)
224. B. Schneier: *Applied Cryptography* (John Wiley & Sons, New York, 1996)
225. M. Langhammer: "Reed-Solomon Codec Design in Programmable Logic," *Communication System Design* (www.csdmag.com) pp. 31–37 (1998)
226. B. Akers: "Binary Decision Diagrams," *IEEE Transactions on Computers* pp. 509–516 (1978)
227. R. Bryant: "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers* pp. 677–691 (1986)
228. A. Sangiovanni-Vincentelli, A. Gamal, J. Rose: "Synthesis Methods for Field Programmable Gate Arrays," *Proceedings of the IEEE* pp. 1057–83 (1993)
229. R. del Rio (1993): "Synthesis of boolean Functions for Field Programmable Gate Arrays," Master's thesis, Univerity of Frankfurt, FB Informatik
230. U. Meyer-Bäse: "Optimal Strategies for Incoherent Demodulation of Narrow Band FM Signals," in *3rd International Symposium on Communication Theory & Applications* (1995), pp. 30–31
231. J. Proakis: *Digital Communications* (McGraw-Hill, New York, 1983)
232. R. Johannesson: "Robustly Optimal One-Half Binary Convolutional Codes," *IEEE Transactions on Information Theory* pp. 464–8 (1975)
233. J. Massey, D. Costello: "Nonsystematic Convolutional Codes for Sequential Decoding in Space Applications," *IEEE Transactions on Communications* pp. 806–813 (1971)
234. F. MacWilliams, J. Sloane: "Pseudo-Random Sequences and Arrays," *Proceedings of the IEEE* pp. 1715–29 (1976)
235. T. Lewis, W. Payne: "Generalized Feedback Shift Register Pseudorandom Number Algorithm," *Journal of the Association for Computing Machinery* pp. 456–458 (1973)
236. P. Bratley, B. Fox, L. Schrage: *A Guide to Simulation* (Springer-Lehrbuch, Heidelberg, 1983), pp. 186–190
237. M. Schroeder: *Number Theory in Science and Communication* (Springer, Heidelberg, 1990)
238. P. Kocher, J. Jaffe, B.Jun: "Differential Power Analysis," in *Lecture Note in Computer Science* (1999), pp. 388–397
239. EFF: *Cracking DES* (O'Reilly & Associates, Sebastopol, 1998), Electronic Frontier Foundation
240. W. Stallings: "Encryption Choices Beyond DES," *Communication System Design* (www.csdmag.com) pp. 37–43 (1998)
241. W. Carter: "FPGAs: Go reconfigure," *Communication System Design* (www.csdmag.com) p. 56 (1998)
242. J. Anderson, T. Aulin, C.E. Sundberg: *Digital Phase Modulation* (Plenum Press, New York, 1986)

243. U. Meyer-Bäse (1989): "Investigation of Thresholdimproving Limiter/Discriminator Demodulator for FM Signals through Computer simulations," Master's thesis, Department of Information Science, Darmstadt University of Technology
244. E. Allmann, T. Wolf (1991): "Design and Implementation of a full digital zero IF Receiver using programmable Gate Arrays and Floatingpoint DSPs," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
245. O. Herrmann: "Quadraturfilter mit rationalem Übertragungsfaktor," *Archiv der elektrischen Übertragung (AEÜ)* pp. 77–84 (1969)
246. O. Herrmann: "Transversalfilter zur Hilbert-Transformation," *Archiv der elektrischen Übertragung (AEÜ)* pp. 581–587 (1969)
247. V. Considine: "Digital Complex Sampling," *Electronics Letters* pp. 608–609 (1983)
248. T.E. Thiel, G.J. Saulnier: "Simplified Complex Digital Sampling Demodulator," *Electronics Letters* pp. 419–421 (1990)
249. U. Meyer-Bäse, W. Hilberg: (1992), "Schmalbandempfänger für Digitalsignale," German patent no. 4219417.2-31
250. B. Schlanske (1992): "Design and Implementation of a Universal Hilbert Sampling Receiver with CORDIC Demodulation for LF FAX Signals using Digital Signal Processor," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
251. A. Dietrich (1992): "Realisation of a Hilbert Sampling Receiver with CORDIC Demodulation for DCF77 Signals using Floatingpoint Signal Processors," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
252. A. Viterbi: *Principles of Coherent Communication* (McGraw-Hill, New York, 1966)
253. F. Gardner: *Phaselock Techniques* (John Wiley & Sons, New York, 1979)
254. H. Geschwinden: *Einführung in die PLL-Technik* (Vieweg, Braunschweig, 1984)
255. R. Best: *Theorie und Anwendung des Phase-locked Loops* (AT Press, Switzerland, 1987)
256. W. Lindsey, C. Chie: "A Survey of Digital Phase-Locked Loops," *Proceedings of the IEEE* pp. 410–431 (1981)
257. R. Sanneman, J. Rowbotham: "Unlock Characteristics of the Optimum Type II Phase-Locked Loop," *IEEE Transactions on Aerospace and Navigational Electronics* pp. 15–24 (1964)
258. J. Stensby: "False Lock in Costas Loops," *Proceedings of the 20th Southeastern Symposium on System Theory* pp. 75–79 (1988)
259. A. Mararios, T. Tozer: "False-Lock Performance Improvement in Costas Loops," *IEEE Transactions on Communications* pp. 2285–88 (1982)
260. A. Makarios, T. Tozer: "False-Look Avoidance Scheme for Costas Loops," *Electronics Letters* pp. 490–2 (1981)
261. U. Meyer-Bäse: "Coherent Demodulation with FPGAs," *Lecture Notes in Computer Science* **1142**, 166–175 (1996)
262. J. Guyot, H. Schmitt (1993): "Design of a full digital Costas Loop using programmable Gate Arrays for coherent Demodulation of Low Frequency Signals," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
263. R. Resch, P. Schreiner (1993): "Design of Full Digital Phase Locked Loops using programmable Gate Arrys for a low Frequency Reciever," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
264. D. McCarty: "Digital PLL Suits FPGAs," *Elektronische Design* p. 81 (1992)

265. J. Holmes: "Tracking-Loop Bias Due to Costas Loop Arm Filter Imbalance," *IEEE Transactions on Communications* pp. 2271–3 (1982)
266. H. Choi: "Effect of Gain and Phase Imbalance on the Performance of Lock Detector of Costas Loop," *IEEE International Conference on Communications, Seattle* pp. 218–222 (1987)
267. N. Wiener: *Extrapolation, Interpolation and Smoothing of Stationary Time Series* (John Wiley & Sons, New York, 1949)
268. S. Haykin: *Adaptive Filter Theory* (Prentice Hall, Englewood Cliffs, New Jersey, 1986)
269. B. Widrow, S. Stearns: *Adaptive Signal Processing* (Prentice Hall, Englewood Cliffs, New Jersey, 1985)
270. C. Cowan, P. Grant: *Adaptive Filters* (Prentice Hall, Englewood Cliffs, New Jersey, 1985)
271. A. Papoulis: *Probability, Random Variables, and Stochastic Processes* (McGraw-Hill, Singapore, 1986)
272. M. Honig, D. Messerschmitt: *Adaptive Filters: Structures, Algorithms, and Applications* (Kluwer Academic Publishers, Norwell, 1984)
273. S. Alexander: *Adaptive Signal Processing: Theory and Application* (Springer, Heidelberg, 1986)
274. N. Shanbhag, K. Parhi: *Pipelined Adaptive Digital Filters* (Kluwer Academic Publishers, Norwell, 1994)
275. B. Mulgrew, C. Cowan: *Adaptive Filters and Equalisers* (Kluwer Academic Publishers, Norwell, 1988)
276. J. Treichler, C. Johnson, M. Larimore: *Theory and Design of Adaptive Filters* (Prentice Hall, Upper Saddle River, New Jersey, 2001)
277. B. Widrow, , J. Glover, J. McCool, J. Kaunitz, C. Williams, R. Hearn, J. Zeidler, E. Dong, R. Goodlin: "Adaptive Noise Cancelling: Principles and Applications," *Proceedings of the IEEE* **63**, 1692–1716 (1975)
278. B. Widrow, J. McCool, M. Larimore, C. Johnson: "Stationary and Nonstationary Learning Characteristics of the LMS Adaptive Filter," *Proceedings of the IEEE* **64**, 1151–1162 (1976)
279. T. Kummura, M. Ikekawa, M. Yoshida, I. Kuroda: "VLIW DSP for Mobile Applications," *IEEE Signal Processing Magazine* **19**, 10–21 (2002)
280. Analog Device: "Application Handbook," 1987
281. L. Horowitz, K. Senne: "Performance Advantage of Complex LMS for Controlling Narrow-Band Adaptive Arrays," *IEEE Transactions on Acoustics, Speech and Signal Processing* **29**, 722–736 (1981)
282. A. Feuer, E. Weinstein: "Convergence Analysis of LMS Filters with Uncorrelated Gaussian Data," *IEEE Transactions on Acoustics, Speech and Signal Processing* **33**, 222–230 (1985)
283. S. Narayan, A. Peterson, M. Narasimha: "Transform Domain LMS Algorithm," *IEEE Transactions on Acoustics, Speech and Signal Processing* **31**, 609–615 (1983)
284. G. Clark, S. Parker, S. Mitra: "A Unified Approach to Time- and Frequency-Domain Realization of FIR Adaptive Digital Filters," *IEEE Transactions on Acoustics, Speech and Signal Processing* **31**, 1073–1083 (1983)
285. F. Beaufays (1995): "Two-Layer Structures for Fast Adaptive Filtering," Ph.D. thesis, Stanford University
286. A. Feuer: "Performance Analysis of Block Least Mean Square Algorithm," *IEEE Transactions on Circuits and Systems* **32**, 960–963 (1985)
287. D. Marshall, W. Jenkins, J. Murphy: "The use of Orthogonal Transforms for Improving Performance of Adaptive Filters," *IEEE Transactions on Circuits and Systems* **36**(4), 499–510 (1989)

288. J. Lee, C. Un: "Performance of Transform-Domain LMS Adaptive Digital Filters," *IEEE Transactions on Acoustics, Speech and Signal Processing* **34**(3), 499–510 (1986)
289. G. Long, F. Ling, J. Proakis: "The LMS Algorithm with Delayed Coefficient Adaption," *IEEE Transactions on Acoustics, Speech and Signal Processing* **37**, 1397–1405 (1989)
290. G. Long, F. Ling, J. Proakis: "Corrections to "The LMS Algorithm with Delayed Coefficient Adaption"," *IEEE Transactions on Signal Processing* **40**, 230–232 (1992)
291. R. Poltmann: "Conversion of the Delayed LMS Algorithm into the LMS Algorithm," *IEEE Signal Processing Letters* **2**, 223 (1995)
292. T. Kimijima, K. Nishikawa, H. Kiya: "An Effective Architecture of Pipelined LMS Adaptive Filters," *IEICE Transactions Fundamentals* **E82-A**, 1428–1434 (1999)
293. D. Jones: "Learning Characteristics of Transpose-Form LMS Adaptive Filters," *IEEE Transactions on Circuits and Systems II* **39**(10), 745–749 (1992)
294. M. Rupp, R. Frenzel: "Analysis of LMS and NLMS Algorithms with Delayed Coefficient Update Under the Presence of Spherically Invariant Processes," *IEEE Transactions on Signal Processing* **42**, 668–672 (1994)
295. M. Rupp: "Saving Complexity of Modified Filtered-X-LMS abd Delayed Update LMS," *IEEE Transactions on Circuits and Systems II* **44**, 57–60 (1997)
296. M. Rupp, A. Sayed: "Robust FxLMS Algorithms with Improved Convergence Performance," *IEEE Transactions on Speech and Audio Processing* **6**, 78–85 (1998)
297. L. Ljung, M. Morf, D. Falconer: "Fast Calculation of Gain Matrices for Recursive Estimation Schemes," *International Journal of Control* **27**, 1–19 (1978)
298. G. Carayannis, D. Manolakis, N. Kalouptsidis: "A Fast Sequential Algorithm for Least-Squares Filtering and Prediction," *IEEE Transactions on Acoustics, Speech and Signal Processing* **31**, 1394–1402 (1983)
299. F. Albu, J. Kadlec, C. Softley, R. Matousek, A. Hermanek, N. Coleman, A. Fagan: "Implementation of (Normalised RLS Lattice on Virtex," *Lecture Notes in Computer Science* **2147**, 91–100 (2001)
300. D. Morales, A. Garcia, E. Castillo, U. Meyer-Baese, A. Palma: "Wavelets for full reconfigurable ECG acquisition system)," in *Proc. SPIE Int. Soc. Opt. Eng.* Orlando (2011), pp. 805 817–1–8
301. M. Keralapura, M. Pourfathi, B. Sikeci-Mergen: "Impact of Contrast Functions in Fast-ICA on Twin ECG Separation," *IAENG International Journal of Computer Science* **38**(1), 1–10 (2011)
302. D. Watkins: *Fundamentals of Matrix Computations* (John Wiley & Sons, New York, 1991)
303. G. Engeln-Müllges, F. Reutter: *Numerisch Mathematik für Ingenieure* (BI Wissenschaftsverlag, Mannheim, 1987)
304. K.K. Shyu, M.H. Li: "FPGA Implementation of FastICA Based on Floating-Point Arithmetic Design for Real-Time Blind Source Separation," in *International Joint Conference on Neural Networks* Vancouver, BC, Canada (2006), pp. 2785–2792
305. Altera: (2008), "QR Matrix Decomposition," application note 506, Ver. 2.0
306. A. Cichocki, R. Unbehauen: *Neural Networks for Optimization and Signal Processing* (John Wiley & Sons, New York, 1993)
307. T. Sanger (1993): "Optimal Unsupervised Learning in Feedforward Neural Networks," Master's thesis, MIT, Dept. of E&C Science

308. Y.Hirai, K. Nishizawa: "Hardware Implementation of the PCA Learning Network by Asynchronous PDM Digital Circuit," in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks* (2000), pp. 65–70
309. S. Kung, K. Diamantaras, J. Taur: "Adaptive Principal Component EXtraction (APEX) and Applications," *IEEE Transactions on Signal Processing* **42**(5), 1202–1216 (1994)
310. B. Yang: "Projection Approximation Subspace Tracking," *IEEE Transactions on Signal Processing* **43**(1), 95–107 (1995)
311. K. Abed-Meraim, A. Chkeif, Y. Hua: "Fast Orthonormal PAST Algorithm," *IEEE Signal Processing Letters* **7**(3), 60–62 (2000)
312. C. Jutten, J. Herault: "Blind Separation of Source, Part I: An Adaptive Algorithm Based on Neuromimetic Architecture," *Signal Processing* **24**(1), 1–10 (1991)
313. A. Hyvaerinen, J. Karhunen, E. Oja: *Independent Component Analysis* (John Wiley & Sons, New York, 2001)
314. A. Cichocki, S. Amari: *Adaptive blind signal and image processing* (John Wiley & Sons, New York, 2002)
315. S. Choi, A. Cichocki, H. Park, S. Lee: "Blind Source Separation and Independent Component Analysis: A Review," *Neural Information Processing - Letters and Reviews* **6**(1), 1–57 (2005)
316. S. Makino: "Blind source separation of convolutive mixtures," in *Proc. SPIE Int. Soc. Opt. Eng.* Orlando (2006), pp. 624709–1–15
317. J. Cardoso, B. Laheld: "Equivariant Adaptive Source Separation," *IEEE Transactions on Signal Processing* **44**(12), 3017–3029 (1996)
318. S. Kim, K. Umeno, R. Takahashi: "FPGA implementation of EASI algorithm," *IEICE Electronics Express* **22**(4), 707–711 (2007)
319. L. Yuan, Z. Sun: "A Survey of Using Sign Function to Improve The Performance of EASI Algorithm," in *Proceedings of the 2007 IEEE International Conference on Mechatronic and Automation* Harbin, China (2007), pp. 2456–2460
320. C. Odom (2013): "Independent Component Analysis Algorithm Fpga Design To Perform Real-Time Blind Source Separation," Master's thesis, Florida State University
321. J. Karhunen, E. Oja, L. Wang, R. Vigario, J. Joutsensalo: "A Class of Neural Networks for Independent Component Analysis," *IEEE Transactions on Neural Networks* **8**(3), 486–504 (1997)
322. A. Hyvarinen, E. Oja: "Independent Component Analysis: Algorithms and Applications," *Neural Networks* **13**(4), 411–430 (2000)
323. A. Hyvarinen, E. Oja: "Independent Component Analysis by general nonlinear Hebbian-like learning rules," *Signal Processing* **64**(1), 301–313 (1998)
324. H. Fastl, E. Zwicker: *Psychoacoustics: Facts and Models* (Springer, Berlin, 2010)
325. ITU-T: (1972), "General Aspects of Digital Transmission Systems," pulse code modulation (PCM) of Voice Frequencies, ITU-T Recommendation G.711
326. W. Chu: *Speech Coding Algorithms: Foundation and Evolution of Standardized Coders* (John Wiley & Sons, New York, 2003)
327. L. Rabiner, R. Schafer: *Theory and Applications of Digital Speech Processing* (Pearson, Upper Saddle River, 2011)
328. IMA: (1992), "Recommended Practices for Enhancing Digital Audio Compatibility in Multimedia Systems," IMA Digital Audio Focus and Technical Working Groups

329. D. Huang: "Lossless Compression for μ -Law (A-Law) and IMA ADPCM on the Basis of a Fast RLS Algorithm," in *IEEE International Conference on Multimedia and Expo* New York (2000), pp. 1775–1778
330. S. Lloyd: "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory* **28**(2), 129137 (1982)
331. D. Wong, B. Juang, A. Gray: "An 800 bit/s Vector Quantization LPC Vocoder," *IEEE Transactions on Acoustics, Speech and Signal Processing* **30**(5), 770–780 (1982)
332. Analog Device: *Digital Signal Processing Applications using the ADSP-2100 family* (Prentice Hall, Englewood Cliffs, New Jersey, 1995), vol. 2
333. S. Aramvith, M. Sun: *Handbook of Image and Video Processing* (Academic Press, New York, 2005), Chap. MPEG-1 and MPEG-2 Video Standards, editor: Al Bovik
334. D. Schulz (1997): "Compression of High Quality Digital Audio Signals using Noiseextraction," Ph.D. thesis, Department of Information Science, Darmstadt University of Technology
335. Xilinx: (2005), "PicoBlaze 8-bit Embedded Microcontroller User Guide," www.xilinx.com
336. V. Heuring, H. Jordan: *Computer Systems Design and Architecture*, 2nd edn. (Prentice Hall, Upper Saddle River, New Jersey, 2004), contribution by M. Murdocca
337. D. Patterson, J. Hennessy: *Computer Organization & Design: The Hardware/Software Interface*, 2nd edn. (Morgan Kaufman Publishers, Inc., San Mateo, CA, 1998)
338. J. Hennessy, D. Patterson: *Computer Architecture: A Quantitative Approach*, 3rd edn. (Morgan Kaufman Publishers, Inc., San Mateo, CA, 2003)
339. M. Murdocca, V. Heuring: *Principles of Computer Architecture*, 1st edn. (Prentice Hall, Upper Saddle River, NJ, 2000)
340. W. Stallings: *Computer Organization & Architecture*, 6th edn. (Prentice Hall, Upper Saddle River, NJ, 2002)
341. R. Bryant, D. O'Hallaron: *Computer Systems: A Programmer's Perspective*, 1st edn. (Prentice Hall, Upper Saddle River, NJ, 2003)
342. C. Rowen: *Engineering the Complex SOC*, 1st edn. (Prentice Hall, Upper Saddle River, NJ, 2004)
343. S. Mazor: "The History of the Microcomputer – Invention and Evolution," *Proceedings of the IEEE* **83**(12), 1601–8 (1995)
344. H. Faggin, M. Hoff, S. Mazor, M. Shima: "The History of the 4004," *IEEE Micro Magazine* **16**, 10–20 (1996)
345. Intel: (2006), "Microprocessor Hall of Fame," <http://www.intel.com/museum>
346. Intel: (1980), "2920 Analog Signal Processor," design handbook
347. TI: (2000), "Technology Innovation," www.ti.com/sc/techinnovations
348. TI: (1983), "TMS3210 Assembly Language Programmer's Guide," digital signal processor products
349. TI: (1993), "TMS320C5x User's Guide," digital signal processor products
350. A. Device: (1993), "ADSP-2103," 3-Volt DSP Microcomputer
351. P. Koopman: *Stack Computers: The New Wave*, 1st edn. (Mountain View Press, La Honda, CA, 1989)
352. Xilinx: (2002), "Creating Embedded Microcontrollers," www.xilinx.com, Part 1–5
353. Altera: (2003), "Nios-32 Bit Programmer's Reference Manual," Nios embedded processor, Ver. 3.1
354. Xilinx: (2002), "Virtex-II Pro," documentation

355. Xilinx: (2005), "MicroBlaze – The Low-Cost and Flexible Processing Solution," www.xilinx.com
356. Altera: (2003), "Nios II Processor Reference Handbook," NII5V-1-5.0
357. B. Parhami: *Computer Architecture: From Microprocessor to Supercomputers*, 1st edn. (Oxford University Press, New York, 2005)
358. Altera: (2004), "Netseminar Nios processor," <http://www.altera.com>
359. A. Hoffmann, H. Meyr, R. Leupers: *Architecture Exploration for Embedded Processors with LISA*, 1st edn. (Kluwer Academic Publishers, Boston, 2002)
360. A. Aho, R. Sethi, J. Ullman: *Compilers: Principles, Techniques, and Tools*, 1st edn. (Addison Wesley Longman, Reading, Massachusetts, 1988)
361. R. Leupers: *Code Optimization Techniques for Embedded Processors*, 2nd edn. (Kluwer Academic Publishers, Boston, 2002)
362. R. Leupers, P. Marwedel: *Retargetable Compiler Technology for Embedded Systems*, 1st edn. (Kluwer Academic Publishers, Boston, 2001)
363. V. Paxson: (1995), "Flex, Version 2.5: A Fast Scanner Generator," <http://www.gnu.org>
364. C. Donnelly, R. Stallman: (2002), "Bison: The YACC-compatible Parser Generator," <http://www.gnu.org>
365. S. Johnson: (1975), "YACC – Yet Another Compiler-Compiler," technical report no. 32, AT&T
366. R. Stallman: (1990), "Using and Porting GNU CC," <http://www.gnu.org>
367. W. Lesk, E. Schmidt: (1975), "LEX – a Lexical Analyzer Generator," technical report no. 39, AT&T
368. T. Niemann: (2004), "A Compact Guide to LEX & YACC," <http://www.epaperpress.com>
369. J. Levine, T. Mason, D. Brown: *lex & yacc*, 2nd edn. (O'Reilly Media Inc., Beijing, 1995)
370. T. Parsons: *Introduction to Compiler Construction*, 1st edn. (Computer Science Press, New York, 1992)
371. A. Schreiner, H. Friedman: *Introduction to Compiler Construction with UNIX*, 1st edn. (Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1985)
372. C. Fraser, D. Hanson: *A Retargetable C Compilers: Design and Implementation*, 1st edn. (Addison-Wesley, Boston, 2003)
373. V. Zivojnovic, J. Velarde, C. Schläger, H. Meyr: "DSPSTONE: A DSP-oriented Benchmarking Methodology," in *International Conference* (19), pp. 1–6
374. Institute for Integrated Systems for Signal Processing: (1994), "DSPstone," final report
375. W. Strauss: "Digital Signal Processing: The New Semiconductor Industry Technology Driver," *IEEE Signal Processing Magazine* pp. 52–56 (2000)
376. Xilinx: (2002), "Virtex-II Pro Platform FPGA," handbook
377. Xilinx: (2005), "Accelerated System Performance with APU-Enhanced Processing," Xcell Journal
378. ARM: (2001), "ARM922T with AHB: Product Overview," <http://www.arm.com>
379. ARM: (2000), "ARM9TDMI Technical Reference Manual," <http://www.arm.com>
380. ARM: (2011), "Cortex-A series processors," <http://www.arm.com>
381. Altera: (2004), "Nios Software Development Reference Manual," <http://www.altera.com>
382. Altera: (2004), "Nios Development Kit, APEX Edition," Getting Started User Guide
383. Altera: (2004), "Nios Development Board Document," <http://www.altera.com>

384. Altera: (2004), "Nios Software Development Tutorial.," <http://www.altera.com>
385. Altera: (2004), "Custom Instruction Tutorial," <http://www.altera.com>
386. B. Fletcher: "FPGA Embedded Processors," in *Embedded Systems Conference* San Francisco, CA (2005), p. 18
387. U. Meyer-Baese, A. Vera, S. Rao, K. Lenk, M. Pattichis: "FPGA Wavelet Processor Design using Language for Instruction-set Architectures (LISA)," in *Proc. SPIE Int. Soc. Opt. Eng.* Orlando (2007), pp. 6576U1–U12
388. D. Sunkara (2004): "Design of Custom Instruction Set for FFT using FPGA-Based Nios processors," Master's thesis, Florida State University
389. U. Meyer-Baese, D. Sunkara, E. Castillo, E.A. Garcia: "Custom Instruction Set NIOS-Based OFDM Processor for FPGAs," in *Proc. SPIE Int. Soc. Opt. Eng.* Orlando (2006), pp. 624801–15
390. J. Ramirez, U. Meyer-Baese, A. Garcia: "Efficient Wavelet Architectures using Field-Programmable Logic and Residue Number System Arithmetic," in *Proc. SPIE Int. Soc. Opt. Eng.* Orlando (2004), pp. 222–232
391. D. Bailey: *Design for Embedded Image Processing on FPGAs*, 1st edn. (John Wiley & Sons, Asia, 2011)
392. W. Pratt: *Digital Image Processing*, 4th edn. (John Wiley & Sons, New York, 2007)
393. R. Gonzalez, R. Woods: *Digital Image Processing*, 2nd edn. (Prentice Hall, New Jersey, 2001)
394. L. Shapiro, G. Stockman: *Computer Vision*, 1st edn. (Prentice Hall, New Jersey, 2001)
395. Y. Wang, J. Ostermann, Y. Zhang: *Video Processing and Communications*, 1st edn. (Prentice Hall, New Jersey, 2001)
396. A. Weeks: *Fundamentals of Electronic Image Processing* (SPIE, US, 1996)
397. J. Bradley: (1994), "XV Interactive Image Display for the X Window System," version 3.10a
398. D. Ziou, S. Tabbone: "Edge Detection Techniques: An Overview," *International Journal Of Pattern Recognition And Image Analysis* **8**(4), 537–559 (1998)
399. ITU: (1992), "T.81 : Information Technology Digital Compression And Coding Of Continuous-Tone Still Images Requirements and Guidelines," CCITT Recommendation T.81
URL <http://www.itu.int/rec/T-REC-T.800-200208-I/en>
400. ITU: (2002), "T.800 : Information technology - JPEG 2000 image coding system: Core coding system," recommendation T.800
URL <http://www.itu.int/rec/T-REC-T.800-200208-I/en>
401. M. Marcellin, M. Gormish, A. Bilgin, M. Boliek: "An overview of JPEG-2000," in *Proceedings Data Compression Conference* (2000), pp. 523–541
402. C. Christopoulos, A. Skodras, T. Ebrahimi: "The JPEG2000 Still Image Coding System: An Overview," *IEEE Transactions on Consumer Electronics* **46**(4), 1103–1127 (2000)
403. T. Acharya, P. Tsai: *JPEG2000 Standard for Image Compression* (John Wiley & Sons, Inc., New Jersey, 2005)
404. B. Fornberg: "Generation of Finite Difference Formulas on Arbitrarily Spaced Grids," *Mathematics Of Computation* **51**(184), 699–706 (1988)
405. J. Prewitt: *Object Enhancement and Extraction* (Academic Press, New York, 1970), Chap. Picture Processing and Psychopictorics, editor: B. Lipkin
406. I. Sobel (1970): "Camera Models and Machine Perception," Ph.D. thesis, Stanford University, Palo Alto, CA

407. I. Abdou, W. Pratt: "Quantitative Design and Evaluation of Enhancement/Thresholding Edge Detectors," *Proceedings of the IEEE* **67**(5), 753–763 (1979)
408. J. Canny: "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence* **8**(6), 679–698 (1986)
409. H. Neoh, A. Hazanchuk: "Adaptive Edge Detection for Real-Time Video Processing using FPGAs," in *Global Signal Processing* Santa Clara Convention Center (2004), pp. 1–6
410. Altera: (2004), "Edge Detection Reference Design," application Note 364, Ver. 1.0
411. Altera: (2007), "Video and Image Processing Design Using FPGAs," white Paper
412. Altera: (2012), "Video IP Cores for Altera DE-Series Boards," ver. 12.0
413. J. Scott, M. Pusateri, M. Mushtaq: "Comparison of 2D median filter hardware implementations for real-time stereo video," in *Applied Imagery Pattern Recognition Workshop* (2008), pp. 1–6
414. H. Eng, K. Ma: "Noise Adaptive Soft-Switching Median Filter," *IEEE Transactions on Image Processing* **10**(2), 242–251 (2001)
415. T. Huang, G. Yang, G. Tang: "A Fast Two-Dimensional Median Filtering Algorithm," *IEEE Transactions on Acoustics, Speech and Signal Processing* **27**(1), 13–18 (1979)
416. H. Hwang, A. Haddad: "Adaptive Median Filters; New Algorithms and Results," *IEEE Transactions on Image Processing* **4**(4), 499–502 (1995)
417. C. Thompson: "The VLSI Complexity of Sorting," *IEEE Transactions on Computers* **32**(12), 1171–1184 (1983)
418. Xilinx: "Two-Dimensional Rank Order Filter," in *Xilinx application note XAPP 953* San Jose (2006)
419. K. Batcher: "Sorting networks and their applications," in *Proceedings of the spring joint computer conference* (ACM, New York, NY, USA, 1968), pp. 307–314
420. G. Bates, S. Nooshabadi: "FPGA implementation of a median filter," in *Proceedings of IEEE Speech and Image Technologies for Computing and Telecommunications IEEE Region 10 Annual Conference* (1997), pp. 437–440
421. K. Benkrid, D. Crookes, A. Benkrid: "Design and implementation of a novel algorithm for general purpose median filtering on FPGAs," in *IEEE International Symposium on Circuits and Systems* Vol. IV (2002), pp. 425–428
422. S. Fahmy, P. Cheung, W. Luk: "Novel Fpga-Based Implementation Of Median And Weighted Median Filters For Image Processing," in *International Conference on Field Programmable Logic and Applications* (2005), pp. 142–147
423. Altera: (2010), "Media Computer System for the Altera DE2-115 Board," ver. 11.0
424. P. Pirsch, N. Demassieux, W. Gehrke: "VLSI Architectures for Video Compression-A Survey," *Proceedings of the IEEE* **83**(2), 220–246 (1995)
425. Y.M.C. Hsueh-Ming Hang, S.C. Cheng: "Motion Estimation for Video Coding Standards," *Journal of VLSI Signal Processing* **17**, 113–136 (1997)
426. M. Ghanbari: "The Cross-Search Algorithm for Motion Estimation," *IEEE Transactions On Communications* **38**(7), 1301–1308 (1990)
427. D. Gonzalez, G. Botella, S. Mookherjee, U. Meyer-Baese, A. Meyer-Baese: "NIOS II processor-based acceleration of motion compensation techniques," in *Proc. SPIE Int. Soc. Opt. Eng., Independent Component Analyses, Wavelets, Neural Networks, Biosystems, and Nanoengineering IX* (2011), pp. 80581C1–12, vol. 8058

428. J. Jain, A. Jain: "Displacement Measurement and Its Application in Inter-frame Image Coding," *IEEE Transactions On Communications* **29**(12), 1799–1808 (1981)
429. T. Koga, K. Iinuma, A. Hirano, Y. Iijima, T. Ishiguro: "Motioncompensated interframe coding for video conferencing," in *Proc. National Telecommunication Conference* New Orleans (1981), pp. C9.6.1–9.6.5
430. R. Kappagantula, K. Rao: "Motion Compensated Interframe Image Prediction," *IEEE Transactions On Communications* **33**(9), 1011–1015 (1985)
431. R. Srinivasan, K. Rao: "Predictive Coding Based on Efficient Motion Estimation," *IEEE Transactions On Communications* **33**(8), 888–896 (1985)
432. A. Puri, H. Hang, D. Schilling: "An Efficient Block-Matching Algorithm For Motion-Compensated Coding," in *IEEE International Conference on Acoustics, Speech, and Signal Processing* (1987), pp. 1063–1066
433. D. Gonzalez, G. Botella, U. Meyer-Baese, C. Garca, C. Sanz, M. Prieto-Matas, F. Tirado: "A Low Cost Matching Motion Estimation Sensor Based on the NIOS II Microprocessor," *Sensors* **12**(10), 13 126–13 149 (2012)
434. D. Gonzalez, G. Botella, A. Meyer-Baese, U. Meyer-Baese: "Optimization of block-matching algorithms unsing custom instruction based paradigm on Nios II microprocessors," in *Proc. SPIE Int. Soc. Opt. Eng., Independent Component Analyses, Wavelets, Neural Networks, Biosystems, and Nanoengineering XI* (2013), pp. 8750Q1–8
435. ITU: (1993), "Line Transmission of non-telephone signals: Video codec for audiovisual services at $p \times 64$ kbits," ITU-T Recommendation H.261
URL <http://www.itu.int/rec/T-REC-H.261/>
436. ITU: (2013), "Advanced video coding for generic audiovisual services," ITU-T Recommendation H.264
URL <http://www.itu.int/rec/T-REC-H.264/>
437. ITU: (1995), "Transmission of non-telephone signals Information technology - Generic coding of moving pictures and associated audio information: Video," ITU-T Recommendation H.262
URL <http://www.itu.int/rec/T-REC-H.262/>
438. ITU: (2005), "Video coding for low bit rate communication," ITU-T Recommendation H.263
URL <http://www.itu.int/rec/T-REC-H.263/>
439. G. Sullivan, J. Ohm, W. Han, T. Wiegand: "Overview of the High Efficiency Video Coding (HEVC) Standard," *IEEE Transactions On Circuits And Systems For Video Technology* **22**(12), 1649–1668 (2012)
440. ITU: (2013), "High efficiency video coding," ITU-T Recommendation H.265
URL <http://www.itu.int/rec/T-REC-H.265/>
441. U. Meyer-Baese: *Digital Signal Processing with Field Programmable Gate Arrays*, 2nd edn. (Springer-Verlag, Berlin, 2004), 527 pages
442. J. Ousterhout: *Tcl and the Tk Toolkit*, 1st edn. (Addison-Wesley, Boston, 1994)
443. M. Harrison, M. McLennan: *Effective Tcl/Tk Programming*, 1st edn. (Addison-Wesley, Reading, Massachusetts, 1998)
444. B. Welch, K. Jones, H. J: *Practical Programming in Tcl and Tk*, 1st edn. (Prentice Hall, Upper Saddle River, NJ, 2003)

Index

- Accumulator 9, 318
- μP 647, 651
- Actel 8
- Adaptive filter 533–629
- Adder
 - binary 80
 - fast carry 80
 - floating-point 115, 125
 - LPM 81, 301
 - pipelined 83
 - size 81
 - speed 81
- ADPCM 618
- A-law 613
- Algorithms
 - Bluestein 424
 - chirp- z 424
 - Cooley–Tukey 442
 - CORDIC 131–141
 - common factor (CFA) 436
 - Goertzel 424
 - Good–Thomas 437
 - fast RLS 586
 - LMS 546, 588
 - prime factor (PFA) 436
 - Rader 427
 - Radix- r 440
 - RLS 575, 588
 - Widrow–Hoff LMS 546
 - Winograd DFT 434
 - Winograd FFT 452
- Altera
 - ARM922T μP 686
 - ARM Cortex-A9 689
- DE2 development board 20
- Devices 8, 11, 22
- Intellectual Property (IP) core 40
 - FFT 458
 - FIR filter 215
 - Image processing blocks 771
 - NCO 40
 - Nios 694
 - Nios II 700, 721, 769, 782
 - PREP test bench 10
 - Quartus II 35
 - Floorplan 36
 - RTL viewer 36
 - Qsys 727
 - TimeQuest 28
 - Timing simulation 38
- AMD 8
- Arbitrary rate conversion 345–374
- Arctan approximation 143
- ARM922T μP 686
- ARM Cortex-A9 689
- Audio compression 613
 - MP3 626
 - MPEG 625
- Bartlett window 189, 419
- Batcher sorting 774
- Bijective 319
- Bison μP tool 661, 672
- Bitreverse 465, 725
- Bit voting 775
- Blackman window 189, 419
- Blowfish 510
- B-spline rate conversion 362

- Butterfly 440, 442
- Canny edge detector 752
- CAST 510
- C compiler 680, 681
- Chebyshev series 142
- Chirp- z algorithm 424
- CIC filter 318–334
 - compensation filter 334
 - RNS design 320
 - interpolator 415
- Coding bounds 481
- Codes
 - block
 - decoders 483
 - encoder 482
 - convolutional
 - comparison 494
 - complexity 493
 - decoder 487, 491
 - encoder 487, 492
 - tree codes 486
- Contour plot 548
- Convergence 546, 547, 549
 - time constant 548
- Convolution
 - Bluestein 466
 - cyclic 465
 - linear 127, 179
- Cooley–Tuckey FFT 442
- CORDIC algorithm 131–141
- cosine approximation 148
- Costas loop
 - architecture 528
 - demodulation 527
 - implementation 529
- CPLD 6, 5
- Cryptography 494–510
- Cumulative histogram 775
- Custom instruction 721, 724
- Cypress 8
- Daubechies 380, 385, 396, 403, 412
- Data encryption standard (DES)
- 503–510
- DCT
 - definition 461
 - fast implementation 464
 - 2D 462
 - JPEG 462
- Decimation 305
- Decimator
 - CIC 321
 - IIR 246
- Demodulator 515
 - Costas loop 527
 - I/Q generation 517
 - zero IF 518
 - PLL 523
- De-noising 405
- DFT
 - definition 418
 - inverse 418
 - filter bank 375
 - Rader 438
 - real 421
 - Winograd 434
- Digital signal processing (DSP) 2, 126
- Dimension reduction 590
- Discrete
 - Cosine transform, *see DCT* 464
 - Fourier transform, *see DFT* 418
 - Hartley transform 468
 - Sine transform (DST) 461
 - Wavelet transform (DWT) 398–403
 - LISA µP 706
- Distributed arithmetic 127–132
 - Optimization
 - Size 131
 - Speed 132
 - signed 204
- Divider 93–109
 - array
 - performance 106
 - size 107
 - convergence 103
 - fast 101

- LPM 106
- nonperforming 99, 171
- nonrestoring 100, 171
- restoring 96
- types 95
- Dyadic DWT 399
- EASI algorithm 605
- Edge detector
 - Canny 752
 - difference of Gaussian 751
 - Laplacian of the Gaussian 750
 - Prewitt 750
 - Sobel 750
- Eigenfrequency 319
- Eigenvalues ratio 552, 558, 559, 582
- Eigenvector 589
 - direct computation 591
 - power method 593
 - Oja learning 594
- Electrocardiogram 592
- Encoder 482, 487, 492
- Error
 - control 475–494
 - cost functions 539
 - residue 542
- Exponential approximation 152
- Farrow rate conversion 358
- Fast RLS algorithm 586
- FFT
 - comparison 456
 - Good–Thomas 437
 - group 440
 - Cooley–Tukey 442
 - in-place 457
 - IP core 458
 - index map 436
 - Nios co-processor 722
 - Radix- r 440
 - rate conversion 347
 - stage 440
 - Winograd 452
- Filter 179–303
 - cascaded integrator comb (CIC) 318–334
 - CIC compensation 334
 - causal 185
 - CSD code 193
 - conjugate mirror 389
 - difference of Gaussian 751
 - distributed arithmetic (DA) 204
 - finite impulse response (FIR) 179–214
 - frequency sampling 342
 - infinite impulse response (IIR) 225–303
 - IP core 215
 - Laplacian of the Gaussian 750
 - lattice 390
 - median 773
 - polyphase implementation 310
 - Prewitt 750
 - signed DA 204
 - Sobel 750
 - symmetric 186
 - 2D 753
 - transposed 181
 - recursive 345
- Filter bank
 - constant
 - bandwidth 395
 - Q 395
 - DFT 375
 - two-channel 380–394
 - aliasing free 383
 - Haar 382
 - lattice 390
 - linear-phase 393
 - lifting 387
 - QMF 380
 - orthogonal 389
 - perfect reconstruction 382
 - polyphase 389
 - mirror frequency 380
 - comparison 394
- Filter design
 - Butterworth 232

- Chebyshev 233
- Comparison of FIR to IIR 226
- elliptic 232
- equiripple 192
- frequency sampling 342
- Kaiser window 188
- Parks–McClellan 191
- Finite impulse response (FIR), *see Filter* 179–214
- Fixed-point arithmetic 106
- Flex µP tool 661, 665
- Flip-flop
 - LPM 17, 33, 83, 83, 83
- Floating-point
 - 754 standard 77, 79
 - addition 115
 - arithmetic 109, 120
 - conversion to fixed-point 111
 - division 116
 - LPM blocks 123
 - multiplication 113
 - numbers 75
 - reciprocal 117
 - rounding 78
 - synthesis results 125
 - VHDL-2008 124
- Floorplan 36
- FPGA
 - Altera’s Cyclone IV E 22
 - architecture 6
 - benchmark 9
 - design compilation 35
 - floor plan 36
 - graphical design entry 36
 - performance analysis 40
 - power dissipation 12
 - registered performance 40
 - routing 5, 25, 26
 - simulation 37
 - size 22, 22
 - technology 8
 - timing 27
 - waveform files 47
 - Xilinx Spartan-6 22
- FPL, *see FPGA and CPLD*
- Fractal 402
- Fractional delay rate conversion 349
- Frequency
 - sampling filter 342
 - synthesizer 32
- Function approximation
 - arctan 143
 - cosine 148
 - Chebyshev series 142
 - exponential 152
 - logarithmic 156
 - sine 148
 - square root 161
 - Taylor series 132
- Galois Field 480
- Gauss primes 73
- General-purpose µP 632, 682
- Generator 71, 72
- Gibb’s phenomenon 188
- Good–Thomas FFT 437
- Goodman/Carey half-band filter 339, 384, 412
- Gradient 545
- H26x 789, 790
- Half-band filter
 - decimator 340
 - factorization 383
 - Goodman and Carey 339, 384
 - definition 339
- Hamming window 189, 419
- Hann window 189, 419
- Harvard µP 652
- Hogenauer filter, *see CIC*
- Homomorphism 318
- IDEA 510
- Identification 537, 551, 560
- Isomorphism 318
- Image
 - Canny edge detector 752
 - compression 462, 743

- edge detection 748
- format 743
- γ correction 746
- JPEG 743
- JPEG 2000 746
- median filter 773
- morphological operations 749
- Independent Component Analysis 601
 - EASI algorithm 605
 - Herault and Jutten method 601
- Index 71
 - multiplier 72
 - maps in FFTs 436
- Infinite impulse response (IIR) filter 225–303
 - finite wordlength effects 239, 255
 - fast filtering using
 - time-domain interleaving 241
 - clustered look-ahead pipelining 243
 - scattered look-ahead pipelining 244
 - decimator design 246
 - parallel processing 247
 - narrow filter
 - BiQuad 261
 - allpass lattice 271,
 - cascade 261
 - lattice WDF 295
 - parallel 265
 - RNS design 250
 - In-place 457
 - Instruction set design 638
 - Intel 633
 - Intellectual Property (IP) core 40
 - FFT 458
 - FIR filter 215
 - Image processing blocks 771
 - NCO 40
 - Interference cancellation 535, 579
 - Interpolation
 - CIC 415
 - *see rate conversion*
 - Inverse
 - multiplicative 437
 - system modeling 536
 - JPEG, *see Image compression*
 - Kaiser
 - window 419
 - window filter design 189
 - Kalman gain 577, 580, 583
 - Kronecker product 452
 - Kurtosis 539
 - Lattice
 - Gray and Markel 279
 - IIR filter 271
 - Semiconductor 7, 13
 - WDF 280
 - Learning curves 551
 - RLS 577, 580
 - Lexical analysis (*see Flex*)
 - LISA μ P 661, 706–722
 - Lifting 387
 - Linear feedback shift register 495
 - LMS algorithm 546, 588
 - normalized 554, 554
 - design 563,
 - pipelined 565
 - delayed 565
 - design 568
 - look-ahead 567
 - transposed 568
 - block FFT 557
 - simplified 573, 574
 - error floor 574
 - Logarithmic approximation 156
 - LPM
 - add_sub 81, 301
 - divider 106, 116
 - multiplier 89
 - RAM 701
 - ROM 48
 - LPC-10e method 625

- MAC 83
- Magnitude 165, 751
- Mean absolute difference 783
- Median filter 773
- MicroBlaze µP 698
- Microprocessor
 - Accumulator 647, 651
 - Bison tool 661, 672
 - C compiler 680, 681
 - DWT 706
 - GPP 632, 682
 - Instruction set design 638
 - Profile 707, 711, 714, 719
 - Intel 633
 - FFT co-processor 721
 - Flex tool 661, 665
 - Lexical analysis (*see Flex*)
 - LISA 661, 706–722
 - Hardcore
 - PowerPC 685
 - ARM922T 686
 - ARM Cortex-A9 689
 - Harvard 652
 - Softcore
 - MicroBlaze 698
 - Nios 694
 - Nios II 700, 721, 769, 782
 - PicoBlaze 632, 690
 - Media processor 777
 - Parser (*see Bison*)
 - PDSP 2, 14, 124, 645, 712
 - RISC 634
 - register file 653
 - Stack 647, 651, 701
 - Super Harvard 652
 - Three address 649, 651
 - Two address 649, 651
 - Vector 716
 - Von-Neuman 652
- μ -law 613
- ModelSim 37, 38
- Moments 539
- MOMS rate conversion 367
- Multiplier
 - adder graph 199, 239
 - array 87
 - block 90
 - Booth 169
 - complex 170, 442
 - FPGA array 88
 - floating-point 113, 125
 - half-square 91
 - index 72
 - LPM 89
 - performance 89
 - QRNS 73
 - quarter square 93, 250
 - size 90
- Modulation 511
 - using CORDIC 514
- Modulo
 - adder 72
 - multiplier 72
 - reconstruction 334
- Motion
 - detection 783
 - vector 784
- MPEG 625, 789, 790
- NAND 5, 47
- NCO IP core 43
- Nios µP 694
- Nios II 700, 721, 769, 782
- Number representation
 - canonical signed digit (CSD) 62, 239
 - diminished by one (D1) 59
 - fixed 106
 - floating-point 75
 - fractional 63, 193
 - one's complement (1C) 59
 - two's complement (2C) 59
 - sign magnitude (SM) 59
- Oja learning 594
- Order filter 180
- Ordering, *see index map*
- Orthogonal

- wavelet transform 385
- filter bank 389

- Parser** (*see* *Bison*)
- Perfect reconstruction** 382
- Phase-locked loop (PLL)**
 - with accumulator reference 518
 - demodulator 524
 - digital 525
 - implementation 525, 526
 - linear 523
- PicoBlaze µP** 632, 690
- Plessey ERA** 5
- Pole/zero diagram** 246, 389
- Polynomial rate conversion** 356
- Polyphase representation** 310, 386
- Power**
 - dissipation 29
 - estimation 554, 554
 - line hum 543, 544, 548, 550, 562, 573
 - method 593
- PowerPC µP** 685
- Prediction** 535
 - forward 583
 - backward 584
- Primitive element** 71
- Programmable signal processor** 2, 14, 124, 645, 712
 - addressing generation 644
- Public key systems** 510
- Principle Component Analysis** 589
 - Sanger's GHA 595

- Quadratic RNS (QRNS)** 73
- Quadrature Mirror Filter (QMF)** 380
- Quartus II** 35
- Qsys** 727

- Rader DFT** 438
- Rate conversion**
 - arbitrary 345–374
 - B-spline 362
 - Farrow 358
- FFT-based** 347
- fractional delay** 349
- MOMS** 367
- polynomial** 356
- rational** 309
- Rational rate conversion** 309
- RC5** 510
- Rectangular window** 189, 419
- Reduced adder graph** 199, 239
- RISC µP** 634
 - register file 653
- RLS algorithm** 575, 580, 586
- RNS**
 - CIC filter 320
 - complex 74
 - IIR filter 250
 - Quadratic 73
 - scaling 334
- ROM**
 - LPM 48
- RSA** 510
- RTL viewer** 36

- Sampling**
 - Frequency 419
 - Time 419
 - *see rate conversion*
- Sea of gates Plessey ERA** 5
- Self-similar** 399
- Sine approximation** 148
- Simulator**
 - ModelSim 37, 38
- Speech compression** 613
 - ADPCM 618
 - A-law 613
 - LPC-10e method 625
- Square root approximation** 161
- Stack µP** 647, 651, 701
- Step size** 549, 550, 559
- Subband filter** 375
- Super Harvard µP** 652
- Symmetry**
 - in filter 186
 - in cryptographic algorithms 510

- Synthesizer
 - accumulator 32
 - PLL with accumulator 518
- Taylor series 132
- Theorem
 - Chinese remainder 71
- Three address μ P 649, 651
- TimeQuest 28
- Timing 27
- Two-channel filter bank 380–394
 - comparison 394
 - lifting 387
 - orthogonal 389
 - QMF 389
 - polyphase 386
- Transformation
 - continuous Wavelet 399
 - discrete cosine 464
 - discrete Fourier 418
 - inverse (IDFT) 418
 - discrete Hartley 468
 - discrete Wavelet 398–403
 - domain LMS 557
 - Fourier 419
 - short-time Fourier (STFT) 395
 - discrete sine 461
- Triple DES 508
- Two address μ P 649, 651
- Vector μ P 716
- Verilog
 - key words 883
- VGA 755
- VHDL
 - styles 17
 - key words 883
- Video processing
 - Motion detection 783
 - standard H26x/MPEG 789, 790
- Von-Neuman μ P 652
- Walsh 531
- Wave digital filter 280
- 3-port 281
- Wavelets 398–403
 - continuous 399
 - de-noising 405
 - linear-phase 393
 - LISA processor 706–722
 - orthogonal 385
- Widrow–Hoff LMS algorithm 546
- Wiener–Hopf equation 542
- Windows 189, 419
- Winograd DFT algorithm 434
- Winograd FFT algorithm 452
- Wordlength
 - IIR filter 239
 - LWDF filter 300
 - narrow band filter 295
- Xilinx
 - ARM Cortex-A9 689
 - Atlys development board 20
 - Devices 8, 11, 21
 - Error correction decoder 479
 - Fast carry adder 80
 - Frequency sampling filter 345
 - FIR filter design 219
 - Hardcore FFTs 457
 - ISIM 32
 - MicroBlaze 698
 - PicoBlaze μ P 632, 690
 - PowerPC Hardcore 685
 - PREP test bench 10
 - Timing simulation 38
 - XBLOCKS 345
- Zech logarithm 72