References

1. B. Dipert: "EDN's first annual PLD directory," *EDN*, pp. 54–84 (2000). http://www.ednmag.com/ednmag/reg/2000/08172000/17cs.htm
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**, 126–148 (1988)
8. J. Donovan: (2002), "The truth about 300 mm," http://www.eet.com
9. Plessey: (1990), "Datasheet," ERA60100
10. J. Greene, E. Hamdy, S. Beal: "Antifuse Field Programmable Gate Arrays," *Proceedings of the IEEE*, pp. 1042–56 (1993)
11. J. Rose, A. Gamal, A. Sangiovanni-Vincentelli: "Architecture of Field-Programmable Gate Arrays," *Proceedings of the IEEE*, pp. 1013–29 (1993)
12. Xilinx: "PREP Benchmark Observations," in *Xilinx-Seminar*, San Jose (1993)
13. Altera: "PREP Benchmarks Reveal FLEX 8000 is Biggest, MAX 7000 is Fastest," in *Altera News & Views*, San Jose (1993)
14. Actel: "PREP Benchmarks Confirm Cost Effectiveness of Field Programmable Gate Arrays," in *Actel-Seminar* (1993)
15. E. Lee: "Programmable DSP Architectures: Part I," *IEEE Transactions on Acoustics, Speech and Signal Processing Magazine*, pp. 4–19 (1988)
16. E. Lee: "Programmable DSP Architectures: Part II," *IEEE Transactions on Acoustics, Speech and Signal Processing Magazine* pp. 4–14 (1989)
17. 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)
18. J. Villasenor, B. Hutchings: "The Flexibility of Configurable Computing," *IEEE Signal Processing Magazine* pp. 67–84 (1998)
19. Xilinx: (1993), "Data book," XC2000, XC3000 and XC4000
20. Altera: (1996), "Datasheet," FLEX 10K CPLD family
21. Altera: (2005), "Cyclone II Device Handbook," Vol. 1
22. F. Vahid: *Embedded System Design* (Prentice Hall, Englewood Cliffs, New Jersey, 1990)
23. J. Hakewill: "Gainin Control over Silicon IP," *Communication Design*, online (2000)

24. 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
25. O. Spaniol: *Computer Arithmetic: Logic and Design* (John Wiley & Sons, New York, 1981)
26. I. Koren: *Computer Arithmetic Algorithms* (Prentice Hall, Englewood Cliffs, New Jersey, 1993)
27. E.E. Swartzlander: *Computer Arithmetic, Vol. I* (Dowden, Hutchingon and Ross, Inc., Stroudsburg, Pennsylvania, 1980), also reprinted by IEEE Computer Society Press 1990
28. E. Swartzlander: *Computer Arithmetic, Vol. II* (IEEE Computer Society Press, Stroudsburg, Pennsylvania, 1990)
29. K. Hwang: *Computer Arithmetic: Principles, Architecture and Design* (John Wiley & Sons, New York, 1979)
30. N. Takagi, H. Yasuura, S. Yajima: "High Speed VLSI multiplication algorithm with a redundant binary addition tree," *IEEE Transactions on Computers* **34** (2) (1985)
31. D. Bull, D. Horrocks: "Reduced-Complexity Digital Filtering Structures using Primitive Operations," *Electronics Letters* pp. 769–771 (1987)
32. D. Bull, D. Horrocks: "Primitive operator digital filters," *IEE Proceedings-G* **138**, 401–411 (1991)
33. 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)
34. 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)
35. F. Taylor, R. Gill, J. Joseph, J. Radke: "A 20 Bit Logarithmic Number System Processor," *IEEE Transactions on Computers* **37** (2) (1988)
36. 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)
37. J. Mitchell: "Computer multiplication and division using binary logarithms," *IRE Transactions on Electronic Computers* **EC-11**, 512–517 (1962)
38. N. Szabo, R. Tanaka: *Residue Arithmetic and its Applications to Computer Technology* (McGraw-Hill, New York, 1967)
39. 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)
40. 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)
41. 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
42. G. Jullien: "Residue Number Scaling and Other Operations Using ROM Arrays," *IEEE Transactions on Communications* **27**, 325–336 (1978)
43. 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

44. G. Zelniker, F. Taylor: "A Reduced-Complexity Finite Field ALU," *IEEE Transactions on Circuits and Systems* **38** (12), 1571–1573 (1991)
45. IEEE: "Standard for Binary Floating-Point Arithmetic," *IEEE Std 754-1985* pp. 1–14 (1985)
46. IEEE: "A Proposed Standard for Binary Floating-Point Arithmetic," *IEEE Transactions on Computers* **14** (12), 51–62 (1981). Task P754
47. 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)
48. M. Bayoumi, G. Jullien, W. Miller: "A VLSI Implementation of Residue Adders," *IEEE Transactions on Circuits and Systems* pp. 284–288 (1987)
49. 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
50. L. Turner, P. Graumann, S. Gibb: "Bit-serial FIR Filters with CSD Coefficients for FPGAs," *Lecture Notes in Computer Science* **975**, 311–320 (1995)
51. J. Logan: "A Square-Summing, High-Speed Multiplier," *Computer Design* pp. 67–70 (1971)
52. Leibowitz: "A Simplified Binary Arithmetic for the Fermat Number Transform," *IEEE Transactions on Acoustics, Speech and Signal Processing* **24**, 356–359 (1976)
53. T. Chen: "A Binary Multiplication Scheme Based on Squaring," *IEEE Transactions on Computers* pp. 678–680 (1971)
54. E. Johnson: "A Digital Quarter Square Multiplier," *IEEE Transactions on Computers* pp. 258–260 (1980)
55. Altera: (2004), "Implementing Multipliers in FPGA Devices," application note 306, Ver. 3.0
56. 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)
57. U. Meyer-Baese: *Digital Signal Processing with Field Programmable Gate Arrays*, 2nd edn. (Springer-Verlag, Berlin, 2004), 527 pages
58. A. Croisier, D. Esteban, M. Levilion, V. Rizo: (1973), "Digital Filter for PCM Encoded Signals," US patent no. 3777130
59. A. Peled, B. Liu: "A New Realization of Digital Filters," *IEEE Transactions on Acoustics, Speech and Signal Processing* **22** (6), 456–462 (1974)
60. K. Yiu: "On Sign-Bit Assignment for a Vector Multiplier," *Proceedings of the IEEE* **64**, 372–373 (1976)
61. K. Kammeyer: "Quantization Error on the Distributed Arithmetic," *IEEE Transactions on Circuits and Systems* **24** (12), 681–689 (1981)
62. F. Taylor: "An Analysis of the Distributed-Arithmetic Digital Filter," *IEEE Transactions on Acoustics, Speech and Signal Processing* **35** (5), 1165–1170 (1986)
63. S. White: "Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review," *IEEE Transactions on Acoustics, Speech and Signal Processing Magazine*, 4–19 (1989)
64. K. Kammeyer: "Digital Filter Realization in Distributed Arithmetic," in *Proc. European Conf. on Circuit Theory and Design* (1976), Genoa, Italy
65. F. Taylor: *Digital Filter Design Handbook* (Marcel Dekker, New York, 1983)
66. H. Nussbaumer: *Fast Fourier Transform and Convolution Algorithms* (Springer, Heidelberg, 1990)
67. H. Schmid: *Decimal Computation* (John Wiley & Sons, New York, 1974)

68. Y. Hu: "CORDIC-Based VLSI Architectures for Digital Signal Processing," *IEEE Signal Processing Magazine* pp. 16–35 (1992)
69. U. Meyer-Bäse, A. Meyer-Bäse, W. Hilberg: "**CO**ordinate **R**otation **DI**gital **C**omputer (CORDIC) Synthesis for FPGA," *Lecture Notes in Computer Science* **849**, 397–408 (1994)
70. J.E. Volder: "The CORDIC Trigonometric computing technique," *IRE Transactions on Electronics Computers* **8** (3), 330–4 (1959)
71. J. Walther: "A Unified algorithm for elementary functions," *Spring Joint Computer Conference* pp. 379–385 (1971)
72. X. Hu, R. Huber, S. Bass: "Expanding the Range of Convergence of the CORDIC Algorithm," *IEEE Transactions on Computers* **40** (1), 13–21 (1991)
73. D. Timmermann (1990): "CORDIC-Algorithmen, Architekturen und monolithische Realisierungen mit Anwendungen in der Bildverarbeitung," Ph.D. thesis, VDI Press, Serie 10, No. 152
74. 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
75. G. Ma (1989): "A Systolic Distributed Arithmetic Computing Machine for Digital Signal Processing and Linear Algebra Applications," Ph.D. thesis, University of Florida, Gainesville
76. Y.H. Hu: "The Quantization Effects of the CORDIC-Algorithm," *IEEE Transactions on Signal Processing* pp. 834–844 (1992)
77. M. Abramowitz, A. Stegun: *Handbook of Mathematical Functions*, 9th edn. (Dover Publications, Inc., New York, 1970)
78. W. Press, W. Teukolsky, W. Vetterling, B. Flannery: *Numerical Recipes in C*, 2nd edn. (Cambridge University Press, Cambrige, 1992)
79. A.V. Oppenheim, R.W. Schafer: *Discrete-Time Signal Processing* (Prentice Hall, Englewood Cliffs, New Jersey, 1992)
80. 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)
81. 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, APCCAS 2006.* (2006), pp. 1555–1558
82. O. Gustafsson, A. Dempster, L. Wanhammer: "Extended Results for Minimum-Adder Constant Integer Multipliers," in *IEEE International Conference on Acoustics, Speech, and Signal Processing* Phoenix (2002), pp. 73–76
83. 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)
84. 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), 1044-1047 (1989)
85. 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)
86. Altera: (2004), "FIR Compiler: MegaCore Function User Guide," ver. 3.1.0
87. R. Hartley: "Subexpression Sharing in Filters Using Canonic Signed Digital Multiplier," *IEEE Transactions on Circuits and Systems II* **30** (10), 677–688 (1996)
88. R. Saal: *Handbook of Filter Design* (AEG-Telefunken, Frankfurt, Germany, 1979)

89. 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)
90. A. Fettweis: "Wave Digital Filters: Theorie and Practice," *Proceedings of the IEEE* pp. 270–327 (1986)
91. R. Crochiere, A. Oppenheim: "Analysis of Linear Digital Networks," *Proceedings of the IEEE* **63** (4), 581–595 (1995)
92. A. Dempster, M. Macleod: "Multiplier blocks and complexity of IIR structures," *Electronics Letters* **30** (22), 1841–1842 (1994)
93. A. Dempster, M. Macleod: "IIR Digital Filter Design Using Minimum Adder Multiplier Blocks," *IEEE Transactions on Circuits and Systems II* **45**, 761–763 (1998)
94. A. Dempster, M. Macleod: "Constant Integer Multiplication using Minimum Adders," *IEE Proceedings - Circuits, Devices & Systems* **141**, 407–413 (1994)
95. 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)
96. H. Loomis, B. Sinha: "High Speed Recursive Digital Filter Realization," *Circuits, Systems, Signal Processing* **3** (3), 267–294 (1984)
97. 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)
98. 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
99. 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)
100. 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)
101. 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
102. P. Vaidyanathan: *Multirate Systems and Filter Banks* (Prentice Hall, Englewood Cliffs, New Jersey, 1993)
103. S. Winograd: "On Computing the Discrete Fourier Transform," *Mathematics of Computation* **32**, 175–199 (1978)
104. Z. Mou, P. Duhamel: "Short-Length FIR Filters and Their Use in Fast Nonrecursive Filtering," *IEEE Transactions on Signal Processing* **39**, 1322–1332 (1991)
105. P. Balla, A. Antoniou, S. Morgera: "Higher Radix Aperiodic-Convolution Algorithms," *IEEE Transactions on Acoustics, Speech and Signal Processing* **34** (1), 60–68 (1986)
106. 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)
107. Harris: (1992), "Datasheet," HSP43220 Decimating Digital Filter
108. Motorola: (1989), "Datasheet," DSPADC16 16–Bit Sigma–Delta Analog–to–Digital Converter

109. 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
110. 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
111. L. Wang, W. Hsieh, T. Truong: "A Fast Computation of 2-D Cubic-Spline Interpolation," *IEEE Signal Processing Letters* **11** (9), 768–771 (2004)
112. T. Laakso, V. Valimaki, M. Karjalainen, U. Laine: "Splitting the Unit Delay," *IEEE Signal Processing Magazine* **13** (1), 30–60 (1996)
113. M. Unser: "Splines: a Perfect Fit for Signal and Image Processing," *IEEE Signal Processing Magazine* **16** (6), 22–38 (1999)
114. 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
115. C. Farrow: "A Continuously Variable Digital Delay Element," in *IEEE International Symposium on Circuits and Systems* Vol. 3 (1988), pp. 2641–2645
116. S. Mitra: *Digital Signal Processing: A Computer-Based Approach*, 3rd edn. (McGraw Hill, Boston, 2006)
117. S. Dooley, R. Stewart, T. Durrani: "Fast On-line B-spline Interpolation," *IEE Electronics Letters* **35** (14), 1130–1131 (1999)
118. Altera: "Farrow-Based Decimating Sample Rate Converter," in *Altera Application Note AN-347* San Jose (2004)
119. 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
120. M. Unser, A. Aldroubi, M. Eden: "B-spline Signal Processing: I – Theory," *IEEE Transactions on Signal Processing* **41** (2), 821–833 (1993)
121. P. Vaidyanathan: "Generalizations of the Sampling Theorem: Seven Decades after Nyquist," *Circuits and Systems I: Fundamental Theory and Applications* **48** (9), 1094–1109 (2001)
122. 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
123. 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)
124. 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)
125. M. Unser, M. Eden: "FIR Approximations of Inverse Filters and Perfect Reconstruction Filter Banks," *Signal Processing* **36** (2), 163–174 (1994)
126. T. Blu, P. Thévenaz, M. Unser: "MOMS: Maximal-Order Interpolation of Minimal Support," *IEEE Transactions on Image Processing* **10** (7), 1069–1080 (2001)
127. T. Blu, P. Thévenaz, M. Unser: "High-Quality Causal Interpolation for Online Unidimenional Signal Processing," in *Proceedings of the Twelfth European Signal Processing Conference (EUSIPCO'04)* (2004), pp. 1417–1420
128. 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
129. 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
130. A. Haar: "Zur Theorie der orthogonalen Funktionensysteme," *Mathematische Annalen* **69**, 331–371 (1910). Dissertation Göttingen 1909
131. 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
132. C. Herley, M. Vetterli: "Wavelets and Recursive Filter Banks," *IEEE Transactions on Signal Processing* **41**, 2536–2556 (1993)
133. I. Daubechies: *Ten Lectures on Wavelets* (Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1992)
134. I. Daubechies, W. Sweldens: "Factoring Wavelet Transforms into Lifting Steps," *The Journal of Fourier Analysis and Applications* **4**, 365–374 (1998)
135. G. Strang, T. Nguyen: *Wavelets and Filter Banks* (Wellesley-Cambridge Press, Wellesley MA, 1996)
136. 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
137. M. Smith, T. Barnwell: "Exact Reconstruction Techniques for Tree-Structured Subband Coders," *IEEE Transactions on Acoustics, Speech and Signal Processing* pp. 434–441 (1986)
138. M. Vetterli, J. Kovacevic: *Wavelets and Subband Coding* (Prentice Hall, Englewood Cliffs, New Jersey, 1995)
139. R. Crochiere, L. Rabiner: *Multirate Digital Signal Processing* (Prentice Hall, Englewood Cliffs, New Jersey, 1983)
140. M. Acheroy, J.M. Mangen, Y. Buhler.: "Progressive Wavelet Algorithm versus JPEG for the Compression of METEOSAT Data," in *SPIE, San Diego* (1995)
141. T. Ebrahimi, M. Kunt: "Image Compression by Gabor Expansion," *Optical Engineering* **30**, 873–880 (1991)
142. D. Gabor: "Theory of communication," *J. Inst. Elect. Eng. (London)* **93**, 429–457 (1946)
143. A. Grossmann, J. Morlet: "Decomposition of Hardy Functions into Square Integrable Wavelets of Constant Shape," *SIAM J. Math. Anal.* **15**, 723–736 (1984)
144. U. Meyer-Bäse: "High Speed Implementation of Gabor and Morlet Wavelet Filterbanks using RNS Frequency Sampling Filters," in *Aerosense 98 SPIE, Orlando* (1998), Vol. 3391, pp. 522–533
145. U. Meyer-Bäse: "Die Hutlets – eine biorthogonale Wavelet-Familie: Effiziente Realisierung durch multipliziererfreie, perfekt rekonstruierende Quadratur Mirror Filter," *Frequenz* pp. 39–49 (1997)
146. 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), Vol. 3078, pp. 670–681
147. 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)

148. C. Burrus: "Index Mappings for Multidimensional Formulation of the DFT and Convolution," *IEEE Transactions on Acoustics, Speech and Signal Processing* **25**, 239–242 (1977)
149. B. Baas: (1998), "SPIFFEE an energy-efficient single-chip 1024-point FFT processor," http://www-star.stanford.edu/ bbaas/fftinfo.html
150. 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
151. TMS: (1996), "TM-66 swiFFT Chip," Texas Memory Systems
152. SHARP: (1997), "BDSP9124 Digital Signal Processor," http://www.butterflydsp.com
153. J. Mellott (1997): "Long Instruction Word Computer," Ph.D. thesis, University of Florida, Gainesville
154. P. Lavoie: "A High-Speed CMOS Implementation of the Winograd Fourier Transform Algorithm," *IEEE Transactions on Signal Processing* **44** (8), 2121–2126 (1996)
155. 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
156. Altera: "Fast Fourier Transform," in *Solution Brief 12*, Altera Corparation (1997)
157. G. Goslin: "Using Xilinx FPGAs to Design Custom Digital Signal Processing Devices," in *Proceedings of the DSP*^{X} (1995), pp. 595–604
158. C. Dick: "Computing 2-D DFTs Using FPGAs," *Lecture Notes in Computer Science: Field-Programmable Logic* pp. 96–105 (1996)
159. S.D. Stearns, D.R. Hush: *Digital Signal Analysis* (Prentice Hall, Englewood Cliffs, New Jersey, 1990)
160. K. Kammeyer, K. Kroschel: *Digitale Signalverarbeitung* (Teubner Studienbücher, Stuttgart, 1989)
161. E. Brigham: *FFT*, 3rd edn. (Oldenbourg Verlag, München Wien, 1987)
162. R. Ramirez: *The FFT: Fundamentals and Concepts* (Prentice Hall, Englewood Cliffs, New Jersey, 1985)
163. R.E. Blahut: *Theory and practice of error control codes* (Addison-Wesley, Melo Park, California, 1984)
164. C. Burrus, T. Parks: *DFT/FFT and Convolution Algorithms* (John Wiley & Sons, New York, 1985)
165. D. Elliott, K. Rao: *Fast Transforms: Algorithms, Analyses, Applications* (Academic Press, New York, 1982)
166. A. Nuttall: "Some Windows with Very Good Sidelobe Behavior," *IEEE Transactions on Acoustics, Speech and Signal Processing* **ASSP-29** (1), 84–91 (1981)
167. U. Meyer-Bäse, K. Damm (1988): "Fast Fourier Transform using Signal Processor," Master's thesis, Department of Information Science, Darmstadt University of Technology
168. 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
169. C. Rader: "Discrete Fourier Transform when the Number of Data Samples is Prime," *Proceedings of the IEEE* **56**, 1107–8 (1968)
170. J. McClellan, C. Rader: *Number Theory in Digital Signal Processing* (Prentice Hall, Englewood Cliffs, New Jersey, 1979)

171. I. Good: "The Relationship between Two Fast Fourier Transforms," *IEEE Transactions on Computers* **20**, 310–317 (1971)
172. L. Thomas: "Using a Computer to Solve Problems in Physics," in *Applications of Digital Computers* (Ginn, Dordrecht, 1963)
173. A. Dandalis, V. Prasanna: "Fast Parallel Implementation of DFT Using Configurable Devices," *Lecture Notes in Computer Science* **1304**, 314–323 (1997)
174. 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), Vol. 3068, pp. 431–442
175. Xilinx: (2000), "High-Performance 256-Point Complex FFT/IFFT," product specification
176. Altera: (2004), "FFT: MegaCore Function User Guide," ver. 2.1.3
177. Z. Wang: "Fast Algorithms for the Discrete W transfrom and for the discrete Fourier Transform," *IEEE Transactions on Acoustics, Speech and Signal Processing* pp. 803–816 (1984)
178. M. Narasimha, A. Peterson: "On the Computation of the Discrete Cosine Transform," *IEEE Transaction on Communications* **26** (6), 934–936 (1978)
179. K. Rao, P. Yip: *Discrete Cosine Transform* (Academic Press, San Diego, CA, 1990)
180. B. Lee: "A New Algorithm to Compute the Discrete Cosine Transform," *IEEE Transactions on Acoustics, Speech and Signal Processing* **32** (6), 1243–1245 (1984)
181. S. Ramachandran, S. Srinivasan, R. Chen: "EPLD-Based Architecture of Real Time 2D-discrete Cosine Transform and Qunatization for Image Compression," in *IEEE International Symposium on Circuits and Systems* Vol. III (1999), pp. 375–378
182. 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)
183. J. Pollard: "The Fast Fourier Transform in a Finite Field," *Mathematics of Computation* **25**, 365–374 (1971)
184. F. Taylor: "An RNS Discrete Fourier Transform Implementation," *IEEE Transactions on Acoustics, Speech and Signal Processing* **38**, 1386–1394 (1990)
185. C. Rader: "Discrete Convolutions via Mersenne Transforms," *IEEE Transactions on Computers* **C-21**, 1269–1273 (1972)
186. N. Bloch: *Abstract Algebra with Applications* (Prentice Hall, Englewood Cliffs, New Jersey, 1987)
187. J. Lipson: *Elements of Algebra and Algebraic Computing* (Addison-Wesley, London, 1981)
188. R. Agrawal, C. Burrus: "Fast Convolution Using Fermat Number Transforms with Applications to Digital Filtering," *IEEE Transactions on Acoustics, Speech and Signal Processing* **22**, 87–97 (1974)
189. W. Siu, A. Constantinides: "On the Computation of Discrete Fourier Transform using Fermat Number Transform," *Proceedings F IEE* **131**, 7–14 (1984)
190. J. McClellan: "Hardware Realization of the Fermat Number Transform," *IEEE Transactions on Acoustics, Speech and Signal Processing* **24** (3), 216–225 (1976)
191. TI: (1993), "User's Guide," TMS320C50, Texas Instruments
192. I. Reed, D. Tufts, X. Yu, T. Truong, M.T. Shih, X. Yin: "Fourier Analysis and Signal Processing by Use of the Möbius Inversion Formula," *IEEE Transactions on Acoustics, Speech and Signal Processing* **38** (3), 458–470 (1990)

193. H. Park, V. Prasanna: "Modular VLSI Arichitectures for Computing the Arithmetic Fourier Transform," *IEEE Transactions on Signal Processing* **41** (6), 2236–2246 (1993)
194. H. Lüke: *Signalübertragung* (Springer, Heidelberg, 1988)
195. D. Herold, R. Huthmann (1990): "Decoder for the Radio Data System (RDS) using Signal Processor TMS320C25," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
196. 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
197. 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
198. U. Meyer-Bäse: "Convolutional Error Decoding with FPGAs," *Lecture Notes in Computer Science* **1142**, 376–175 (1996)
199. R. Watzel (1993): "Design of Paging Scheme and Implementation of the Suitable Cryto-Controller using FPGAs," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
200. J. Maier, T. Schubert (1993): "Design of Convolutional Decoders using FPGAs for Error Correction in a Paging System," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
201. U. Meyer-Bäse et al.: "Zum bestehenden Übertragungsprotokoll kompatible Fehlerkorrektur," in *Funkuhren Zeitsignale Normalfrequenzen* (1993), pp. 99–112
202. 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
203. P. Sweeney: *Error Control Coding* (Prentice Hall, New York, 1991)
204. D. Wiggert: *Error-Control Coding and Applications* (Artech House, Dedham, Mass., 1988)
205. G. Clark, J. Cain: *Error-Correction Coding for Digital Communications* (Plenum Press, New York, 1988)
206. W. Stahnke: "Primitive Binary Polynomials," *Mathematics of Computation* pp. 977–980 (1973)
207. W. Fumy, H. Riess: *Kryptographie* (R. Oldenbourg Verlag, München, 1988)
208. B. Schneier: *Applied Cryptography* (John Wiley & Sons, New York, 1996)
209. M. Langhammer: "Reed-Solomon Codec Design in Programmable Logic," *Communication System Design (www.csdmag.com)* pp. 31–37 (1998)
210. B. Akers: "Binary Decusion Diagrams," *IEEE Transactions on Computers* pp. 509–516 (1978)
211. R. Bryant: "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers* pp. 677–691 (1986)
212. A. Sangiovanni-Vincentelli, A. Gamal, J. Rose: "Synthesis Methods for Field Programmable Gate Arrays," *Proceedings of the IEEE* pp. 1057–83 (1993)
213. R. del Rio (1993): "Synthesis of boolean Functions for Field Programmable Gate Arrays," Master's thesis, Univerity of Frankfurt, FB Informatik
214. 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
215. J. Proakis: *Digital Communications* (McGraw-Hill, New York, 1983)
216. R. Johannesson: "Robustly Optimal One-Half Binary Convolutional Codes," *IEEE Transactions on Information Theory* pp. 464–8 (1975)

217. J. Massey, D. Costello: "Nonsystematic Convolutional Codes for Sequential Decoding in Space Applications," *IEEE Transactions on Communications* pp. 806–813 (1971)
218. F. MacWilliams, J. Sloane: "Pseudo-Random Sequences and Arrays," *Proceedings of the IEEE* pp. 1715–29 (1976)
219. T. Lewis, W. Payne: "Generalized Feedback Shift Register Pseudorandom Number Algorithm," *Journal of the Association for Computing Machinery* pp. 456–458 (1973)
220. P. Bratley, B. Fox, L. Schrage: *A Guide to Simulation* (Springer-Lehrbuch, Heidelberg, 1983), pp. 186–190
221. M. Schroeder: *Number Theory in Science and Communication* (Springer, Heidelberg, 1990)
222. P. Kocher, J. Jaffe, B.Jun: "Differential Power Analysis," in *Lecture Notes in Computer Science* (1999), pp. 388–397, www.cryptography.com
223. EFF: *Cracking DES* (O'Reilly & Associates, Sebastopol, 1998), Electronic Frontier Foundation
224. W. Stallings: "Encryption Choices Beyond DES," *Communication System Design (www.csdmag.com)* pp. 37–43 (1998)
225. W. Carter: "FPGAs: Go reconfigure," *Communication System Design (www.csdmag.com)* p. 56 (1998)
226. J. Anderson, T. Aulin, C.E. Sundberg: *Digital Phase Modulation* (Plenum Press, New York, 1986)
227. 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
228. 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
229. O. Herrmann: "Quadraturfilter mit rationalem Übertragungsfaktor," *Archiv der elektrischen Übertragung (AEÜ)* pp. 77–84 (1969)
230. O. Herrmann: "Transversalfilter zur Hilbert-Transformation," *Archiv der elektrischen Übertragung (AEÜ)* pp. 581–587 (1969)
231. V. Considine: "Digital Complex Sampling," *Electronics Letters* pp. 608–609 (1983)
232. T.E. Thiel, G.J. Saulnier: "Simplified Complex Digital Sampling Demodulator," *Electronics Letters* pp. 419–421 (1990)
233. U. Meyer-Bäse, W. Hilberg: (1992), "Schmalbandempfänger für Digitalsignale," German patent no. 4219417.2-31
234. 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
235. 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
236. A. Viterbi: *Principles of Coherent Communication* (McGraw-Hill, New York, 1966)
237. F. Gardner: *Phaselock Techniques* (John Wiley & Sons, New York, 1979)
238. H. Geschwinde: *Einführung in die PLL-Technik* (Vieweg, Braunschweig, 1984)
239. R. Best: *Theorie und Anwendung des Phase-locked Loops* (AT Press, Schwitzerland, 1987)

240. W. Lindsey, C. Chie: "A Survey of Digital Phase-Locked Loops," *Proceedings of the IEEE* pp. 410–431 (1981)
241. 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)
242. J. Stensby: "False Lock in Costas Loops," *Proceedings of the 20th Southeastern Symposium on System Theory*, pp. 75–79 (1988)
243. A. Mararios, T. Tozer: "False-Lock Performance Improvement in Costas Loops," *IEEE Transactions on Communications* pp. 2285–88 (1982)
244. A. Makarios, T. Tozer: "False-Look Avoidance Scheme for Costas Loops," *Electronics Letters* pp. 490–2 (1981)
245. U. Meyer-Bäse: "Coherent Demodulation with FPGAs," *Lecture Notes in Computer Science* **1142**, 166–175 (1996)
246. 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
247. 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
248. D. McCarty: "Digital PLL Suits FPGAs," *Elektronic Design* p. 81 (1992)
249. J. Holmes: "Tracking-Loop Bias Due to Costas Loop Arm Filter Imbalance," *IEEE Transactions on Communications* pp. 2271–3 (1982)
250. 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)
251. N. Wiener: *Extrapolation, Interpolation and Smoothing of Stationary Time Series* (John Wiley & Sons, New York, 1949)
252. S. Haykin: *Adaptive Filter Theory* (Prentice Hall, Englewood Cliffs, New Jersey, 1986)
253. B. Widrow, S. Stearns: *Adaptive Signal Processing* (Prentice Hall, Englewood Cliffs, New Jersey, 1985)
254. C. Cowan, P. Grant: *Adaptive Filters* (Prentice Hall, Englewood Cliffs, New Jersey, 1985)
255. A. Papoulis: *Probability, Random Variables, and Stochastic Processes* (McGraw–Hill, Singapore, 1986)
256. M. Honig, D. Messerschmitt: *Adaptive Filters: Structures, Algorithms, and Applications* (Kluwer Academic Publishers, Norwell, 1984)
257. S. Alexander: *Adaptive Signal Processing: Theory and Application* (Springer, Heidelberg, 1986)
258. N. Shanbhag, K. Parhi: *Pipelined Adaptive Digital Filters* (Kluwer Academic Publishers, Norwell, 1994)
259. B. Mulgrew, C. Cowan: *Adaptive Filters and Equalisers* (Kluwer Academic Publishers, Norwell, 1988)
260. J. Treichler, C. Johnson, M. Larimore: *Theory and Design of Adaptive Filters* (Prentice Hall, Upper Saddle River, New Jersey, 2001)
261. 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)
262. 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)

263. T. Kummura, M. Ikekawa, M. Yoshida, I. Kuroda: "VLIW DSP for Mobile Applications," *IEEE Signal Processing Magazine* **19**, 10–21 (2002)
264. Analog Device: "Application Handbook," 1987
265. 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)
266. 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)
267. S. Narayan, A. Peterson, M. Narasimha: "Transform Domain LMS Algorithm," *IEEE Transactions on Acoustics, Speech and Signal Processing* **31**, 609–615 (1983)
268. 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)
269. F. Beaufays (1995): "Two-Layer Structures for Fast Adaptive Filtering," Ph.D. thesis, Stanford University
270. A. Feuer: "Performance Analysis of Block Least Mean Square Algorithm," *IEEE Transactions on Circuits and Systems* **32**, 960–963 (1985)
271. 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)
272. 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)
273. 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)
274. G. Long, F. Ling, J. Proakis: "Corrections to "The LMS Algorithm with Delayed Coefficient Adaption"," *IEEE Transactions on Signal Processing* **40**, 230–232 (1992)
275. R. Poltmann: "Conversion of the Delayed LMS Algorithm into the LMS Algorithm," *IEEE Signal Processing Letters* **2**, 223 (1995)
276. T. Kimijima, K. Nishikawa, H. Kiya: "An Effective Architecture of Pipelined LMS Adaptive Filters," *IEICE Transactions Fundamentals* **E82-A**, 1428–1434 (1999)
277. D. Jones: "Learning Characteristics of Transpose-Form LMS Adaptive Filters," *IEEE Transactions on Circuits and Systems II* **39** (10), 745–749 (1992)
278. M. Rupp, R. Frenzel: "Analysis of LMS and NLMS Algorithms with Delayed Coefficient Update Under the Presence of Spherically Invariant Processess," *IEEE Transactions on Signal Processing* **42**, 668–672 (1994)
279. M. Rupp: "Saving Complexity of Modified Filtered-X-LMS abd Delayed Update LMS," *IEEE Transactions on Circuits and Systems II* **44**, 57–60 (1997)
280. M. Rupp, A. Sayed: "Robust FxLMS Algorithms with Improved Convergence Performance," *IEEE Transactions on Speech and Audio Processing* **6**, 78–85 (1998)
281. L. Ljung, M. Morf, D. Falconer: "Fast Calculation of Gain Matrices for Recursive Estimation Schemes," *International Journal of Control* **27**, 1–19 (1978)
282. 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)

283. 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)
284. Xilinx: (2005), "PicoBlaze 8-bit EMbedded Microcontroller User Guide," www.xilinx.com
285. V. Heuring, H. Jordan: *Computer Systems Design and Architecture*, 2nd edn. (Prentice Hall, Upper Saddle RIver, New Jersey, 2004), contribution by M. Murdocca
286. D. Patterson, J. Hennessy: *Computer Organization & Design: The Hardware/Software Interface*, 2nd edn. (Morgan Kaufman Publishers, Inc., San Mateo, CA, 1998)
287. J. Hennessy, D. Patterson: *Computer Architecture: A Quantitative Approach*, 3rd edn. (Morgan Kaufman Publishers, Inc., San Mateo, CA, 2003)
288. M. Murdocca, V. Heuring: *Principles of Computer Architecture*, 1st edn. (Prentice Hall, Upper Saddle River, NJ, 2000), jAVA machine overview
289. W. Stallings: *Computer Organization & Architecture*, 6th edn. (Prentice Hall, Upper Saddle River, NJ, 2002)
290. R. Bryant, D. O'Hallaron: *Computer Systems: A Programmer's Perspective*, 1st edn. (Prentice Hall, Upper Saddle River, NJ, 2003)
291. C. Rowen: *Engineering the Complex SOC*, 1st edn. (Prentice Hall, Upper Saddle River, NJ, 2004)
292. S. Mazor: "The History of the Microcomputer – Invention and Evolution," *Proceedings of the IEEE* **83** (12), 1601–8 (1995)
293. H. Faggin, M. Hoff, S. Mazor, M. Shima: "The History of the 4004," *IEEE Micro Magazine* **16**, 10–20 (1996)
294. Intel: (2006), "Microprocessor Hall of Fame," http://www.intel.com/museum
295. Intel: (1980), "2920 Analog Signal Processor," design handbook
296. TI: (2000), "Technology Inovation," www.ti.com/sc/techinnovations
297. TI: (1983), "TMS3210 Assembly Language Programmer's Guide," digital signal processor products
298. TI: (1993), "TMS320C5x User's Guide," digital signal processing products
299. Analog Device: (1993), "ADSP-2103," 3-Volt DSP Microcomputer
300. P. Koopman: *Stack Computers: The New Wave*, 1st edn. (Mountain View Press, La Honda, CA, 1989)
301. Xilinx: (2002), "Creating Embedded Microcontrollers," www.xilinx.com, Part 1-5
302. Altera: (2003), "Nios-32 Bit Programmer's Reference Manual," Nios embedded processor, Ver. 3.1
303. Xilinx: (2002), "Virtex-II Pro," documentation
304. Xilinx: (2005), "MicroBlaze – The Low-Cost and Flexible Processing Solution," www.xilinx.com
305. Altera: (2003), "Nios II Processor Reference Handbook," NII5V-1-5.0
306. B. Parhami: *Computer Architecture: From Microprocessor to Supercomputers*, 1st edn. (Oxford University Press, New York, 2005)
307. Altera: (2004), "Netseminar Nios processor," http://www.altera.com
308. A. Hoffmann, H. Meyr, R. Leupers: *Architecture Exploration for Embedded Processors with LISA*, 1st edn. (Kluwer Academic Publishers, Boston, 2002)
309. A. Aho, R. Sethi, J. Ullman: *Compilers: Principles, Techniques, and Tools*, 1st edn. (Addison Wesley Longman, Reading, Massachusetts, 1988)
310. R. Leupers: *Code Optimization Techniques for Embedded Processors*, 2nd edn. (Kluwer Academic Publishers, Boston, 2002)
311. R. Leupers, P. Marwedel: *Retargetable Compiler Technology for Embedded Systems*, 1st edn. (Kluwer Academic Publishers, Boston, 2001)

312. V. Paxson: (1995), "Flex, Version 2.5: A Fast Scanner Generator," http://www.gnu.org
313. C. Donnelly, R. Stallman: (2002), "Bison: The YACC-Compatible Parser Generator," http://www.gnu.org
314. S. Johnson: (1975), "YACC – Yet Another Compiler-Compiler," technical report no. 32, AT&T
315. R. Stallman: (1990), "Using and Porting GNU CC," http://www.gnu.org
316. W. Lesk, E. Schmidt: (1975), "LEX – a Lexical Analyzer Generator," technical report no. 39, AT&T
317. T. Niemann: (2004), "A Compact Guide to LEX & YACC," http://www.epaperpress.com
318. J. Levine, T. Mason, D. Brown: *lex & yacc*, 2nd edn. (O'Reilly Media Inc., Beijing, 1995)
319. T. Parsons: *Intorduction to Compiler Construction*, 1st edn. (Computer Science Press, New York, 1992)
320. A. Schreiner, H. Friedman: *Introduction to Compiler Construction with UNIX*, 1st edn. (Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1985)
321. C. Fraser, D. Hanson: *A Retargetable C Compilers: Design and Implementation*, 1st edn. (Addison-Wesley, Boston, 2003)
322. V. Zivojnovic, J. Velarde, C. Schläger, H. Meyr: "DSPSTONE: A DSP-orioented Benchmarking Methodology," in *International Conference* (19), pp. 1–6
323. Institute for Integrated Systems for Signal Processing: (1994), "DSPstone," final report
324. W. Strauss: "Digital Signal Processing: The New Semiconductor Industry Technology Driver," *IEEE Signal Processing Magazine* pp. 52–56 (2000)
325. Xilinx: (2002), "Virtex-II Pro Platform FPGA," handbook
326. Xilinx: "Accelerated System Performance with APU-Enhanced Processing," *Xcell Journal* pp. 1-4, first quarter (2005)
Xilinx: (2007), "Virtex-4 Online Documentation," http://www.xilinx.com
327. ARM: (2001), "ARM922T with AHB: Product Overview," http://www.arm.com
328. ARM: (2000), "ARM9TDMI Technical Reference Manual," http://www.arm.com
329. Altera: (2004), "Nios Software Development Reference Manual," http://www.altera.com
330. Altera: (2004), "Nios Development Kit, APEX Edition," Getting Started User Guide
331. Altera: (2004), "Nios Development Board Document," http://www.altera.com
332. Altera: (2004), "Nios Software Development Tutorial," http://www.altera.com
333. Altera: (2004), "Custom Instruction Tutorial," http://www.altera.com
334. B. Fletcher: "FPGA Embedded Processors," in *Embedded Systems Conference* San Francisco, CA (2005), p. 18, www.memec.com
335. 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), Vol. 6576, pp. 6576U1–U12
336. D. Sunkara (2004): "Design of Custom Instruction Set for FFT using FPGA-Based Nios processors," Master's thesis, FSU
337. 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), Vol. 6248, pp. 6248o01–15

338. 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), Vol. 5439, pp. 222–232
339. J. Bhasker: *Verilog HDL Synthesis* (Start Galaxy Publishing, Allentown, PA, 1998)
340. IEEE: (1995), "Standard Hardware Description Language Based on the Verilog Hardware Description Language," language reference manual std 1364-1995
341. IEEE: (2001), "Standard Verilog Hardware Description Language," language reference manual std 1364-2001
342. S. Sutherland: "The IEEE Verilog 1364-2001 Standard: Whats New, and Why You Need It," in *Proceedings 9th Annual International HDL Conference and Exhibition* Santa Clara, CA (2000), p. 8, http://www.sutherland-hdl.com
343. Xilinx: (2004), "Verilog-2001 Support in XST," XST version 6.1 help
344. Altera: (2004), "Quartus II Support for Verilog 2001," Quartus II version 4.2 help
345. Synopsys: (2003), "Common VCS and HDL Compiler (Presto Verilog) 2001 Constructs," SolvNet doc id: 002232
346. J. Ousterhout: *Tcl and the Tk Toolkit*, 1st edn. (Addison-Wesley, Boston, 1994)
347. M. Harrison, M. McLennan: *Effective Tcl/Tk Programming*, 1st edn. (Addison-Wesley, Reading, Massachusetts, 1998)
348. B. Welch, K. Jones, H. J: *Practical Programming in Tcl and Tk*, 1st edn. (Prentice Hall, Upper Saddle River, NJ, 2003)

A. Verilog Source Code 2001

The first and second editions of the book include Verilog using the IEEE 1364-1995 standard [340]. This third edition takes advantage of several improvements that are documented in the IEEE 1364-2001 standard, which is available for about $100 from the IEEE bookstore [341, 342]. The Verilog 2001 improvements have now found implementations in all major design tools, [343, 344, 345] and we want to briefly to review the most important new features that are used. We only review the implemented new features; for all the new features see [341].

- The entity description in the 1364-1995 standard requires that (similar to the Kernighan and Ritchie `C` coding) all ports appear twice, first in the port list and then in the port data-type description, e.g.,

```
module iir_pipe (x_in, y_out, clk); //----> Interface

  parameter W = 14; // Bit width - 1
  input          clk;
  input  [W:0]  x_in;   // Input
  output [W:0]  y_out;  // Result
  ...
```

  Note that all ports (`x_in`, `y_out`, and `clk`) are defined twice. In the 1364-2001 standard (see Sect. 12.3.4 LRM) this duplication is no longer required, i.e., the new coding is done as follows:

```
module iir_pipe      //----> Interface
 #(parameter W = 14) // Bit width - 1
  (input          clk,
   input  signed [W:0]  x_in,   // Input
   output signed [W:0]  y_out);  // Result
```

- Signed data types are available in the 1364-2001 standard, which allows arithmetic signed operations to be simplified. In the signal definition line the signed keyword is introduced after the input, output, reg or wire keywords, e.g.,

```
reg signed [W:0] x, y;
```

Conversion between signed and unsigned type can be accomplished via the $signed or $unsigned conversion functions, see Sect. 4.5 LRM. For signed constants we introduce a small s *or* capital S between the hyphen and the base, e.g., `'sd90` for a signed constant 90. Signed arithmetic operations can be done using the conventional divide / or multiply * operators. For power-of-2 factors we can use the new arithmetic left `<<<` or right shift `>>>` operations, see Sect. 4.1.23 LRM. Note the three shift operations symbols used to distinguished to the unsigned shifts that use two shift symbols. Signed or zero extension is automatically done depending on the data type. From the IIR filter examples we can now replace the old-style operation:

```
...
y  <= x + {y[W],y[W:1]} + {{2{y[W]}},y[W:2]};
...                              // i.e., x + y / 2 + y / 4;
```

with the 1364-2001 Verilog style operations using the divide operator:

```
y <= x + y / 'sd2 + y / 'sd4;  // div with / uses 92 LEs.
```

Note the definition as signed divide for the constants, the code

```
    y <= x + y / 2 + y / 4;
```

will show incorrect simulation results in Quartus II in versions 4.0, 4.2, 5.0, 5.1 but works fine in our web edition 6.0. Alteratively we can use the arithmetic right shift operator to implement the divide by power-of-2 values.

```
y <= x + (y >>> 1) + (y >>> 2);// div with >>> uses 60 LEs
```

It is evident that this notation makes the arithmetic operation much more readable than the old-style coding. Although both operations are functional equivalent, the Quartus synthesis results reveals that the divide is mapped to a different architecture and needs therefore more LEs (92 compared with 60 LEs) than used by the arithmetic shift operations. From the comparison to the VHDL synthesis data we conclude that 60 LEs is the expect result. This is the reason we will use the arithmetic left and right shift throughout the examples.

- The implicit `event_expression` list allows one to add automatically all right-hand-side variables to be added to the event expression, i.e., from the `bfproc` examples

```
...
 always @(Are or Bre or Aim or Bim)
  begin
  ...
```

we now simply use

```
...
  always @(*)
  begin
  ...
```

This reduces essentially the RTL simulation errors due to missing variables in the event listing, see Sect. 9.7.5 LRM for details.

- The `generate` statement introduced in the 1364-2001 Verilog standard allows one to instantiate several components using a single generate loop construct. The LRM Sects. 12.1.3.2-4 show eight different generate examples. We use the generate in the `fir_gen`, `fir_lms`, and `fir6dlms` files. Note that you also need to define a `genvar` as a loop variable used for the generate statement, see Sect. 12.1.3.1 LRM for details.

The 1364-2001 Verilog standard introduces 21 new keywords. We use the new keywords `endgenerate`, `generate`, `genvar`, `signed`, and `unsigned`. We have not used the new keywords:

```
automatic, cell, config, design, endconfig, incdir, include,
instance, liblist, library, localparam, noshowcancelled,
pulsestyle_onevent, pulsestyle_ondetect, showcancelled, use
```

The next pages contain the Verilog 1364-2001 code of all design examples. The old style Verilog 1364-1995 code can be found in [57]. The synthesis results for the examples are listed on page 731.

```
//************************************************************
// IEEE STD 1364-2001 Verilog file: example.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//************************************************************
//`include "220model.v"   // Using predefined components

module example   //----> Interface
  #(parameter WIDTH =8)    // Bit width
 (input  clk,
  input  [WIDTH-1:0] a, b, op1,
  output [WIDTH-1:0] sum, d);

  wire [WIDTH-1:0]  c;         // Auxiliary variables
  reg  [WIDTH-1:0]  s;         // Infer FF with always
  wire [WIDTH-1:0] op2, op3;

  wire  clkena, ADD, ena, aset, sclr, sset, aload, sload,
                aclr, ovf1, cin1; // Auxiliary lpm signals

// Default for add:
  assign cin1=0; assign aclr=0; assign ADD=1;
```

```
  assign ena=1; assign aclr=0; assign aset=0;
  assign sclr=0; assign sset=0; assign aload=0;
  assign sload=0; assign clkena=0; // Default for FF

  assign op2 = b;       // Only one vector type in Verilog;
            // no conversion int -> logic vector necessary

// Note when using 220model.v ALL component's signals
// must be defined, default values can only be used for
// the parameters.

  lpm_add_sub add1          //----> Component instantiation
  ( .result(op3), .dataa(op1), .datab(op2)); // Used ports
//  .cin(cin1),.cout(cr1), .add_sub(ADD), .clken(clkena),
//  .clock(clk), .overflow(ovl1), .aclr(aclr)); // Unused
    defparam add1.lpm_width = WIDTH;
    defparam add1.lpm_representation = "SIGNED";

  lpm_ff reg1
  ( .data(op3), .q(sum), .clock(clk));  // Used ports
//  .enable(ena), .aclr(aclr), .aset(aset), .sclr(sclr),
//  .sset(sset), .aload(aload), .sload(sload)); // Unused
    defparam reg1.lpm_width = WIDTH;

  assign c = a + b; //----> Continuous assignment statement

  always @(posedge clk)  //----> Behavioral style
  begin : p1             // Infer register
    s = c + s;           // Signal assignment statement
  end
  assign d = s;

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
//`include "220model.v"

module fun_text             //----> Interface
  #(parameter WIDTH = 32)   // Bit width
 (input          clk,
```

```
  input  [WIDTH-1:0]  M,
  output [7:0]  sin, acc);

  wire [WIDTH-1:0] s, acc32;
  wire [7:0]    msbs;               // Auxiliary vectors
  wire  ADD, ena, aset, sclr, sset; // Auxiliary signals
  wire aload, sload, aclr, ovf1, cin1, clkena;

  // Default for add:
  assign clkena=0; assign cin1=0; assign ADD=1;
  //default for FF:
  assign ena=1; assign aclr=0; assign aset=0;
  assign sclr=0; assign sset=0; assign aload=0;
  assign sload=0;

  lpm_add_sub add_1                        // Add M to acc32
  ( .result(s), .dataa(acc32), .datab(M)); // Used ports
//  .cout(cr1), .add_sub(ADD), .overflow(ovl1),  // Unused
//  .clock(clk),.cin(cin1), .clken(clkena), .aclr(aclr));
//
    defparam add_1.lpm_width = WIDTH;
    defparam add_1.lpm_representation = "UNSIGNED";


  lpm_ff reg_1                                 // Save accu
  ( .data(s), .q(acc32), .clock(clk));        // Used ports
//  .enable(ena), .aclr(aclr), .aset(aset), // Unused ports
//  .sset(sset), .aload(aload), .sload(sload),.sclr(sclr));
    defparam reg_1.lpm_width = WIDTH;

  assign msbs = acc32[WIDTH-1:WIDTH-8];
  assign acc  = msbs;

  lpm_rom rom1
  ( .q(sin), .inclock(clk), .outclock(clk),
                            .address(msbs)); // Used ports
//                     .memenab(ena) ) ;      // Unused port
    defparam rom1.lpm_width = 8;
    defparam rom1.lpm_widthad = 8;
    defparam rom1.lpm_file = "sine.mif";

endmodule

//*********************************************************
// IEEE STD 1364-2001 Verilog file: cmul7p8.v
```

```
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*********************************************************
module cmul7p8                      // ------> Interface
        (input  signed [4:0] x,
         output signed [4:0] y0, y1, y2, y3);

  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
//*********************************************************
//`include "220model.v"

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,  // 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 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

//`include "220model.v"

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,  // Single bits
  input   clk);              // 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 stages
// uses no components

//`include "220model.v"

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] l0, l1, r0, v0, s0;    // LSBs of inputs
  reg [WIDTH0:0] q0;                      // LSBs of inputs
  reg [WIDTH1-1:0] l2, l3, r1, s1;      // 2. LSBs of input
  reg [WIDTH1:0] v1, q1;                // 2. LSBs of input
  reg [WIDTH2-1:0] l4, l5, s2, h7;          // 2. MSBs bits
  reg [WIDTH2:0] q2, v2, r2;                // 2. MSBs bits
  reg [WIDTH3-1:0] l6, l7, 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
  l0[WIDTH0-1:0] <= x[WIDTH0-1:0];
  l1[WIDTH0-1:0] <= y[WIDTH0-1:0];
  // Split 2. LSBs from input x,y
  l2[WIDTH1-1:0] <= x[WIDTH1-1+WIDTH0:WIDTH0];
  l3[WIDTH1-1:0] <= y[WIDTH1-1+WIDTH0:WIDTH0];
  // Split 2. MSBs from input x,y
  l4[WIDTH2-1:0] <= x[WIDTH2-1+WIDTH01:WIDTH01];
  l5[WIDTH2-1:0] <= y[WIDTH2-1+WIDTH01:WIDTH01];
  // Split MSBs from input x,y
  l6[WIDTH3-1:0] <= x[WIDTH3-1+WIDTH012:WIDTH012];
  l7[WIDTH3-1:0] <= y[WIDTH3-1+WIDTH012:WIDTH012];

//************* First stage of the adder  *****************
  q0 <= {1'b0, l0} + {1'b0, l1};  // Add LSBs of x and y
  q1 <= {1'b0, l2} + {1'b0, l3};  // Add 2. LSBs of x / y
  q2 <= {1'b0, l4} + {1'b0, l5};  // Add 2. MSBs of x/y
  q3 <= l6 + l7;                  // 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: mul_ser.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//************************************************************
module mul_ser      //----> Interface
 (input         clk, reset,
  input  signed [7:0]  x,
  input   [7:0]  a,
  output reg signed [15:0] y);

  always @(posedge clk) //-> Multiplier in behavioral style
  begin : States
    parameter s0=0, s1=1, s2=2;
    reg [2:0] count;
    reg [1:0] s;                    // FSM state register
    reg signed [15:0] p, t;         // Double bit width
    reg  [7:0] a_reg;

    if (reset)              // Asynchronous reset
```

```
      s <= s0;
    else
      case (s)
        s0 : begin         // Initialization step
          a_reg <= a;
          s <= s1;
          count = 0;
          p <= 0;      // Product register reset
          t <= x;      // Set temporary shift register to x
        end
        s1 : begin          // Processing step
          if (count == 7)   // Multiplication ready
            s <= s2;
          else
            begin
            if (a_reg[0] == 1) // Use LSB for bit select
              p <= p + t;      // Add 2^k
            a_reg <= a_reg >>> 1;
            t <= t <<< 1;
            count = count + 1;
            s <= s1;
          end
        end
        s2 : begin       // Output of result to y and
          y <= p;        // start next multiplication
          s <= s0;
        end
      endcase
  end

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
//         Numerator / Denominator = Quotient and Remainder
// OR:       Numerator = Quotient * Denominator + Remainder

module div_res(
  input         clk, reset,
  input  [7:0] n_in,
  input  [5:0] d_in,
```

```
output reg [5:0] r_out,
output reg [7:0] q_out);

parameter s0=0, s1=1, s2=2, s3=3; // State assignments

// Divider in behavioral style
always @(posedge clk or posedge reset)
begin : F // Finite state machine
  reg [3:0] count;
  reg [1:0] s;          // FSM state
  reg  [13:0] d;        // Double bit width unsigned
  reg  signed [13:0] r; // Double bit width signed
  reg  [7:0] q;

  if (reset)              // Asynchronous reset
    s <= s0;
  else
    case (s)
      s0 : begin         // Initialization step
        s <= s1;
        count = 0;
        q <= 0;           // Reset quotient register
        d <= d_in << 7;   // Load aligned denominator
        r <= n_in;        // Remainder = numerator
      end
      s1 : begin         // Processing step
        r <= r - d;       // Subtract denominator
        s <= s2;
      end
      s2 : begin          // Restoring step
        if (r < 0) begin  // Check r < 0
          r <= r + d;     // Restore previous remainder
          q <= q << 1;     // LSB = 0 and SLL
          end
        else
          q <= (q << 1) + 1; // LSB = 1 and SLL
        count = count + 1;
        d <= d >> 1;

        if (count == 8)   // Division ready ?
          s <= s3;
        else
          s <= s1;
      end
```

```
        s3 : begin       // Output of result
          q_out <= q[7:0];
          r_out <= r[5:0];
          s <= s0;   // 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
//         Numerator / Denominator = Quotient and Remainder
// OR:       Numerator = Quotient * Denominator + Remainder

module div_aegp
 (input         clk, reset,
  input  [8:0] n_in,
  input  [8:0] d_in,
  output reg [8:0] q_out);

  always @(posedge clk or posedge reset) //-> Divider in
  begin : States                         // behavioral style
    parameter s0=0, s1=1, s2=2;
    reg [1:0] count;
    reg [1:0] state;
    reg [9:0] x, t, f;        // one guard bit
    reg [17:0] tempx, tempt;

    if (reset)              // Asynchronous reset
      state <= s0;
    else
      case (state)
        s0 : begin              // Initialization step
          state <= s1;
          count = 0;
          t <= {1'b0, d_in};    // Load denominator
          x <= {1'b0, n_in};    // Load numerator
        end
        s1 : begin            // Processing step
```

```
            f = 512 - t;        // TWO - t
            tempx = (x * f);  // Product in full
            tempt = (t * f);  // bitwidth
            x <= tempx >> 8;  // Factional 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,
 input  signed [W:0] x_in, y_in,
 output reg signed [W:0] r, phi, eps);


// There is bit access in Quartus array types
// in Verilog 2001, therefore use single vectors
// but use a separate line 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 clk) begin  //----> Infer registers
    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

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];
  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

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,
   input signed [W-1:0] x_in,
   //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,
   output reg signed [W-1:0] f_out);

  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) begin
    x <= x_in;     // FF for input and output
    f_out <= f;
  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 coeffcients-1
            parameter W= 17) // -- Bitwidth -1
       (input clk,
        input signed [W:0] x_in,
        output reg signed [W:0] f_out);

  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)
  begin : Store
    x <= x_in;     // Store input in register
  end

  always @(posedge clk)        // 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_out  <= 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, reset,
  input  [16:0]  x_in,
  output [16:0]  a_o, imm_o, f_o,
  output reg [2:0]  ind_o,
  output reg [1:0]  count_o,
  output [16:0]  x_o,pre_o,post_o,
  output reg [16:0]  f_out);

  // 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 [2:0] s, op;
  reg [16:0] x; // Auxilary
  reg signed [16:0] a, b, f, imm; // ALU data
  reg [16:0] pre, post;
  // 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
  reg signed [3:0] ind;
  reg [1:0] count;

  if (reset)                    // Asynchronous reset
    s <= start;
  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 == 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;
      end                    // start next cycle
    endcase
  end
  ind_o <= ind;
  count_o <= count;
end

always @(posedge clk) // Define the ALU operations
begin : ALU
```

```
      case (op)
          load    : f  <= imm;
          mac     : f  <= (a * f / 32768) + imm;
          scale   : f  <= a * f;
          denorm  : f  <= (a * f /32768);
          nop     : f  <= f;
          default : f  <= f;
      endcase
    end

    always @*
    begin : EXP
      reg [16:0] slv;
      reg [16:0] po, pr;
      integer K, L;

      slv = x_in;
      // Compute pre-scaling:
      for (K=0; K <= 15; K= K+1)
        if (slv[K] == 1)
          L <= K;
      pre = 1 << (14-L);
      // 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;
```

```
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
            Mpipe = 3) // Pipeline steps of multiplier
 (input clk, Load_x,  // std_logic
  input signed [W1-1:0] x_in, c_in,  // Inputs
  output signed [W4-1:0] y_out);  // Results

  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

  wire  signed [W2-1:0] sum;  // Auxilary signals
  wire  clken, aclr;

  assign sum=0; assign aclr=0; // Default for mult
  assign clken=0;

//----> Load Data or Coefficient
  always @(posedge clk)
    begin: Load
    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 begin
      x <= x_in; // Get one data sample at a time
    end
```

```
  end

//----> Compute sum-of-products
  always @(posedge clk)
    begin: SOP
  // Compute the transposed filter additions
    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
  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 pipelined multiplier
  lpm_mult mul_I            // Multiply  x*c[I] = p[I]
    (.clock(clk), .dataa(x), .datab(c[I]), .result(p[I]));
//  .sum(sum), .clken(clken), .aclr(aclr)); // Unused ports
    defparam mul_I.lpm_widtha = W1;
    defparam mul_I.lpm_widthb = W1;
    defparam mul_I.lpm_widthp = W2;
    defparam mul_I.lpm_widths = W2;
    defparam mul_I.lpm_pipeline = Mpipe;
    defparam mul_I.lpm_representation = "SIGNED";
  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,
  input signed [7:0] x,
  output reg signed [7:0] y);

// Tapped delay line array of bytes
  reg  signed  [7:0] tap [0:3];
// For bit access use single vectors in Verilog
```

```
  integer I;

  always @(posedge clk)  //----> Behavioral style
  begin : p1
   // Compute output y with the filter coefficients weight.
   // The coefficients are [-1  3.75  3.75  -1].
   // Multiplication and division for Altera MaxPlusII can
   // be done in Verilog 2001 with signed shifts !
    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

endmodule

//*********************************************************
// IEEE STD 1364-2001 Verilog file: dafsm.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*********************************************************
`include "case3.v" // User-defined component

module dafsm        //--> Interface
 (input         clk, reset,
  input  [2:0]  x_in0, x_in1, x_in2,
  output [2:0]  lut,
  output reg [5:0]  y);

  reg    [2:0]  x0, x1, x2;
  wire   [2:0]  table_in, table_out;

  reg [5:0] p;  // temporary register

  assign table_in[0] = x0[0];
  assign table_in[1] = x1[0];
  assign table_in[2] = x2[0];

  always @(posedge clk or posedge reset)
  begin : DA                 //----> DA in behavioral style
    parameter s0=0, s1=1;
    reg [0:0] state;
```

```
    reg [1:0] count;   // Counts the shifts

    if (reset)              // Asynchronous reset
      state <= s0;
    else
    case (state)
      s0 : begin       // Initialization
        state <= s1;
        count = 0;
        p  <= 0;
        x0 <= x_in0;
        x1 <= x_in1;
        x2 <= x_in2;
      end
      s1 : begin                 // Processing step
        if (count == 3) begin    // Is sum of product done?
          y <= p;              // Output of result to y and
          state <= s0;         // start next sum of product
        end
        else begin
          p <= (p >> 1) + (table_out << 2); // p/2+table*4
          x0[0] <= x0[1];
          x0[1] <= x0[2];
          x1[0] <= x1[1];
          x1[1] <= x1[2];
          x2[0] <= x2[1];
          x2[1] <= x2[2];
          count = count + 1;
          state <= s1;
        end
      end
    endcase
  end

  case3 LC_Table0
  ( .table_in(table_in), .table_out(table_out));

  assign lut = table_out; // Provide test signal

endmodule

//*******************************************************
// IEEE STD 1364-2001 Verilog file: case3.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*******************************************************
```

```
module case3
  (input  [2:0] table_in,  // Three bit
  output reg [2:0] table_out); // Range 0 to 6

// 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 =  5;
      4 :     table_out =  1;
      5 :     table_out =  3;
      6 :     table_out =  4;
      7 :     table_out =  6;
      default : ;
    endcase
  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;
        4  : table0out01 = 14;
```

```
      5  : table0out01 = 15;
      6  : table0out01 = 17;
      7  : table0out01 = 18;
      8  : table0out01 = 16;
      9  : table0out01 = 17;
      10 : table0out01 = 19;
      11 : table0out01 = 20;
      12 : table0out01 = 21;
      13 : table0out01 = 22;
      14 : table0out01 = 24;
      15 : table0out01 = 25;
      default ;
    endcase
  end

endmodule

//*********************************************************
// IEEE STD 1364-2001 Verilog file: darom.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*********************************************************
//`include "220model.v"

module darom  //--> Interface
 (input         clk, reset,
  input  [2:0]  x_in0, x_in1, x_in2,
  output [2:0]  lut,
  output reg [5:0]  y);

  reg    [2:0]  x0, x1, x2;
  wire   [2:0]  table_in, table_out;

  reg [5:0] p;  // Temporary register
  wire ena;

  assign ena=1;

  assign table_in[0] = x0[0];
  assign table_in[1] = x1[0];
  assign table_in[2] = x2[0];

  always @(posedge clk or posedge reset)
  begin : DA                 //----> DA in behavioral style
    parameter s0=0, s1=1;
    reg [0:0] state;
```

```
    reg [1:0] count;   // Counts the shifts

    if (reset)         // Asynchronous reset
      state <= s0;
    else
      case (state)
        s0 : begin    // Initialization
          state <= s1;
          count = 0;
          p  <= 0;
          x0 <= x_in0;
          x1 <= x_in1;
          x2 <= x_in2;
        end
      s1 : begin                  // Processing step
          if (count == 3) begin  // Is sum of product done?
            y <= (p >> 1) + (table_out << 2);// Output to y
            state <= s0;   // and start next sum of product
          end
          else begin
            p <= (p >> 1) + (table_out << 2);
            x0[0] <= x0[1];
            x0[1] <= x0[2];
            x1[0] <= x1[1];
            x1[1] <= x1[2];
            x2[0] <= x2[1];
            x2[1] <= x2[2];
            count = count + 1;
            state <= s1;
          end
        end
        default : ;
      endcase
  end

  lpm_rom rom_1     // Used ports:
  ( .outclock(clk),.address(table_in), .q(table_out));
// .inclock(clk),  .memenab(ena)); // Unused
    defparam rom_1.lpm_width = 3;
    defparam rom_1.lpm_widthad = 3;
    defparam rom_1.lpm_outdata = "REGISTERED";
    defparam rom_1.lpm_address_control = "UNREGISTERED";
    defparam rom_1.lpm_file = "darom3.mif";
```

```
  assign lut = table_out; // Provide test signal

endmodule

//************************************************************
// IEEE STD 1364-2001 Verilog file: dasign.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//************************************************************
`include "case3s.v" // User-defined component

module dasign              //-> Interface
 (input         clk, reset,
  input signed  [3:0]  x_in0, x_in1, x_in2,
  output [3:0]  lut,
  output reg signed [6:0]  y);

  reg  signed  [3:0] x0, x1, x2;
  wire signed  [2:0] table_in;
  wire signed  [3:0] table_out;

  reg [6:0] p;  // Temporary register

  assign table_in[0] = x0[0];
  assign table_in[1] = x1[0];
  assign table_in[2] = x2[0];

  always @(posedge clk or posedge reset)// DA in behavioral
  begin : DA                                          // style
    parameter s0=0, s1=1;
    integer k;
    reg [0:0] state;
    reg [2:0] count;           // Counts the shifts

    if (reset)                 // Asynchronous reset
      state <= s0;
    else
      case (state)
        s0 : begin             // Initialization step
          state <= s1;
          count = 0;
          p  <= 0;
          x0 <= x_in0;
          x1 <= x_in1;
          x2 <= x_in2;
        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,
  input signed [3:0]  x_in,
  output reg signed[6:0]  y);

  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)  //----> DA in behavioral style
  begin : DA
    integer k,l;
    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[l][k] <= x[l][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
```

```
  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  signed [W:0] x_in,  // Input
        output signed [W:0] y_out,  // Result
        input         clk);

  reg signed [W:0] x, y;

// initial begin
//  y=0;
//  x=0;
// end

// Use FFs for input and recursive part
always @(posedge clk) begin    // Note: there is a signed
  x  <= x_in;                  // integer in Verilog 2001
  y  <= x + (y >>> 1) + (y >>> 2); // >>> uses fewer LEs

//y  <= x + y / 2 + y / 4; // div with / uses more LEs

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,
  input signed [W:0]  x_in,     // Input
  output signed [W:0]  y_out);  // Result

  reg signed [W:0] x, x3, sx;
  reg signed [W:0] y, y9;

  always @(posedge clk)  // Infer FFs for input, output and
  begin                  // pipeline stages;
    x   <= x_in;         // use nonblocking FF assignments
    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

  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, reset,
  input signed [W:0]  x_in,
  output signed [W:0]  y_out,
  output         clk2);

  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;

  always @(posedge clk)          // Clock divider by 2
  begin : clk_divider            // for input clk
     clk_div2 <= ! clk_div2;
  end

  always @(posedge clk)          // Split x into even
```

```
  begin : Multiplex              // and odd samples;
    parameter even=0, odd=1;     // recombine y at clk rate
    reg [0:0] state;

    if (reset)                   // Asynchronous reset
      state <= even;
    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;

  always @(negedge clk_div2)
  begin: Arithmetic
    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

endmodule

//*******************************************************
// IEEE STD 1364-2001 Verilog file: cic3r32.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
```

```
//************************************************************
module cic3r32  //----> Interface
 (input        clk, reset,
  input signed [7:0] x_in,
  output signed [9:0] y_out,
  output reg clk2);

  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) // 3 integrator sections
  begin : Int
      x    <= x_in;
      i0   <= i0 + x;
      i1   <= i1 + i0 ;
      i2   <= i2 + i1 ;
   end

  always @(posedge clk) // 3 comb sections
```

```
  begin : Comb
    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, reset,
  output reg clk2,
  input  signed [7:0] x_in,
  output signed [9:0] y_out);

  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) // 3 integrator sections
  begin : Int
      x   <= x_in;
      i0  <= i0 + x;
      i1  <= i1 + i0[25:5];
      i2  <= i2 + i1[20:5];
  end

  always @(posedge clk) // 3 comb sections
  begin : Comb
    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: db4poly.v
```

```
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*********************************************************
module db4poly     //----> Interface
 (input          clk, reset,
  output         clk2,
  input signed [7:0]   x_in,
  output signed [16:0]  x_e, x_o, g0, g1, // Test signals
  output signed [8:0]   y_out);

  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)              // Asynchronous reset
      state <= even;
    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 @(negedge clk_div2) // Infer registers;
  begin : AddPolyphase      // use nonblocking assignments
//---------- 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

// 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: 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, reset,  // Clock + reset for the registers
  input signed [7:0] x_in,
  output [3:0] count_o,
  output ena_in_o, ena_out_o, ena_io_o,
  output signed [8:0] f0_o, f1_o, f2_o,
```

```
output signed [8:0] y_out);

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 begin
    if (count == 11)
      count <= 0;
    else
      count <= count + 1;
  end
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)       //----> Input delay line
begin :  INPUTMUX
  integer I;    // loop variable

  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)       //----> Output delay line
begin  : OUPUTMUX
  integer I;    // loop variable

  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=OL; I>=1; I=I-1)
      obuf[I] <= obuf[I-1];       // shift one
  end
end

always @(posedge clk)      //----> One tapped delay line
begin : TAP                // get 4 samples at one time
  integer I;    // loop variable

  if (ena_io) begin
  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) // Compute sum-of-products for f0
  begin :  SOP0
    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

    f0 <= sum >>> 8;
  end

 always @(posedge clk) // Compute sum-of-products for f1
  begin :  SOP1
    f1 <= x[5];  // No scaling, i.e., unit inpulse
  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

      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, reset,  // Clock/reset for the registers
       input signed [7:0] x_in,
       output [3:0] count_o,
       output ena_in_o, ena_out_o,
       output signed [8:0] c0_o, c1_o, c2_o, c3_o,
       output [8:0] d_out,
       output reg signed [8:0] y_out);

  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

always @(posedge clk)
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)      //----> One tapped delay line
begin : TAP
  integer I;    // loop variable

  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)
begin : GET                // Get 4 samples at one time
  integer I;    // loop variable

  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) // Compute sum-of-products for f0
  begin :  SOP
    reg signed [8:0] y; // 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 (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;
      y = c2 + ((c3 * d) >>> 8); // d is scale by 256
      y = ((y * d) >>> 8) + c1;
      y = ((y * d) >>> 8) + c0;

      y_out <= y; // 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, reset,  // Clock/reset for registers
        input signed [7:0] x_in,
        output [3:0] count_o,
        output ena_in_o, ena_out_o,
        output signed [8:0] c0_o, c1_o, c2_o, c3_o, xiir_o,
        output signed [8:0] y_out);


  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)      //----> Behavioral Style
  begin : IIR  // Compute iir coefficients first
    reg signed [8:0] x1;    // x * 1

    if (ena_in) begin
      xiir <= (3 * x1 >>> 1) - (xiir >>> 1);
      x1 = x_in;
    end
  end

  always @(posedge clk)      //----> One tapped delay line
  begin : TAP
    integer I;    // loop variable

    if (ena_in) begin
      for (I=1; I<=IL; I=I+1)
        ibuf[I-1] <= ibuf[I];   // Shift one

      ibuf[IL] <= xiir;         // Input in register IL
```

```
      end
    end

    always @(posedge clk)      //----> One tapped delay line
    begin : GET                // get 4 samples at one time
      integer I;    // loop variable

      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) // Compute sum-of-products for f0
    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 (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, reset,
  output         clk2,
  input signed  [7:0]  x_in,
  output signed [16:0]  x_e, x_o,
  output reg signed [8:0]   g, h);

  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)              // Asynchronous reset
      state <= even;
    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

  //******** Multipy 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 @(negedge clk_div2)
    begin: LowerTreeFF
        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 @(negedge clk_div2)
    begin: OutputScale
      g <= up1 >>> 8;      // i.e. up1 / 256
      h <= low1 >>> 8;     // i.e. low1 / 256;
    end

  endmodule

  //*********************************************************
```

```
// IEEE STD 1364-2001 Verilog file: rader7.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*********************************************************
module rader7           //---> Interface
 (input         clk, reset,
  input  [7:0]  x_in,
  output reg signed [10:0] y_real, y_imag);

  reg signed [10:0]  accu;              // Signal for X[0]
 // Direct bit access of 2D vector in Quartus Verilog 2001
 // possible 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]

  always @(posedge clk or posedge reset)   // State machine
  begin : States                          // for RADER filter
    parameter Start=0, Load=1, Run=2;
    reg [1:0] state;
    reg [4:0] count;

    if (reset)               // Asynchronous reset
      state <= Start;
    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
          state <= Start;      // Output of result
        end                    // and start again
        else begin
          y_real  <= (re[0] >>> 8) + x_0;
                                 // i.e. re[0]/256+x[0]
          y_imag  <= (im[0] >>> 8);     // i.e. im[0]/256
          state <= Run;
        end
        count <= count + 1;
      end
    endcase
end

always @(posedge clk)    // Structure of the two FIR
begin : Structure        // filters in transposed form
  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

always @(posedge clk)     //  Note that all signals
begin : Coeffs            //  are globally defined
// Compute the filter coefficients and use FFs
```

```
    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

  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: ccmul.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*********************************************************
//`include "220model.v"

module ccmul #(parameter W2 = 17,   // Multiplier bit width
                         W1 = 9,    // Bit width c+s sum
                         W  = 8)    // Input bit width
 (input clk,  // Clock for the output register
  input signed [W-1:0] x_in, y_in, c_in,  // Inputs
  input signed [W1-1:0]  cps_in, cms_in,  // Inputs
  output reg signed [W-1:0]    r_out, i_out);  // Results


  wire signed [W-1:0] x, y, c ;       // Inputs and outputs
  wire signed [W2-1:0] r, i, cmsy, cpsx, xmyc, sum; //Prod.
  wire signed [W1-1:0] xmy, cps, cms, sxtx, sxty;//x-y etc.


  wire  clken, cr1, ovl1, cin1, aclr, ADD, SUB;
                                       // Auxiliary signals
  assign cin1=0; assign aclr=0; assign ADD=1; assign SUB=0;
  assign cr1=0; assign sum=0; assign clken=0;
                                         // Default for add
```

```
  assign x   = x_in;   // x
  assign y   = y_in;   // j * y
  assign c   = c_in;   // cos
  assign cps = cps_in; // cos + sin
  assign cms = cms_in; // cos - sin

  always @(posedge clk) begin
    r_out <= r[W2-3:W-1];      // Scaling and FF for output
    i_out <= i[W2-3:W-1];
  end

//********* ccmul with 3 mul. and 3 add/sub  **************
  assign sxtx  = x;     // Possible growth for
  assign sxty  = y;     // sub_1 -> sign extension

  lpm_add_sub sub_1                  // Sub:  x - y
  ( .result(xmy), .dataa(sxtx), .datab(sxty));// Used ports
//  .add_sub(SUB), .cout(cr1), .overflow(ovl1), .cin(cin1),
//   .clken(clken), .clock(clk), .aclr(aclr));  // Unused
    defparam sub_1.lpm_width = W1;
    defparam sub_1.lpm_representation = "SIGNED";
    defparam sub_1.lpm_direction = "sub";

  lpm_mult mul_1               // Multiply  (x-y)*c = xmyc
  ( .dataa(xmy), .datab(c), .result(xmyc)); // Used ports
//  .sum(sum), .clock(clk), .clken(clken), .aclr(aclr));
                                          // Unused ports
    defparam mul_1.lpm_widtha = W1;
    defparam mul_1.lpm_widthb = W;
    defparam mul_1.lpm_widthp = W2;
    defparam mul_1.lpm_widths = W2;
    defparam mul_1.lpm_representation = "SIGNED";

  lpm_mult mul_2                // Multiply (c-s)*y = cmsy
  ( .dataa(cms), .datab(y), .result(cmsy)); // Used ports
//  .sum(sum), .clock(clk), .clken(clken), .aclr(aclr));
                                          // Unused ports
    defparam mul_2.lpm_widtha = W1;
    defparam mul_2.lpm_widthb = W;
    defparam mul_2.lpm_widthp = W2;
    defparam mul_2.lpm_widths = W2;
    defparam mul_2.lpm_representation = "SIGNED";

  lpm_mult mul_3                // Multiply (c+s)*x = cpsx
```

```
  ( .dataa(cps), .datab(x), .result(cpsx)); // Used ports
//  .sum(sum), .clock(clk), .clken(clken), .aclr(aclr));
                                            // Unused ports
    defparam mul_3.lpm_widtha= W1;
    defparam mul_3.lpm_widthb = W;
    defparam mul_3.lpm_widthp = W2;
    defparam mul_3.lpm_widths = W2;
    defparam mul_3.lpm_representation = "SIGNED";

  lpm_add_sub add_1        // Add:  r <= (x-y)*c + (c-s)*y
  ( .dataa(cmsy), .datab(xmyc), .result(r));  // Used ports
//  .add_sub(ADD), .cout(cr1), .overflow(ovl1), .cin(cin1),
//  .clken(clken), .clock(clk), .aclr(aclr));  // Unused
    defparam add_1.lpm_width = W2;
    defparam add_1.lpm_representation = "SIGNED";
    defparam add_1.lpm_direction = "add";

  lpm_add_sub sub_2          // Sub: i <= (c+s)*x - (x-y)*c
  ( .dataa(cpsx), .datab(xmyc), .result(i));  // Used ports
// .add_sub(SUB), .cout(cr1), .overflow(ovl1), .clock(clk),
//  .cin(cin1),  .clken(clken), .aclr(aclr));  // Unused
    defparam sub_2.lpm_width = W2;
    defparam sub_2.lpm_representation = "SIGNED";
    defparam sub_2.lpm_direction = "sub";

endmodule

//*********************************************************
// IEEE STD 1364-2001 Verilog file: bfproc.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*********************************************************
//`include "220model.v"
//`include "ccmul.v"

module bfproc #(parameter W2 = 17,  // Multiplier bit width
                          W1 = 9,    // Bit width c+s sum
                          W  = 8)    // Input bit width
 (input clk,  // Clock for the output register
  input signed [W-1:0] Are_in, Aim_in,      // 8-bit inputs
  input signed [W-1:0] Bre_in, Bim_in, c_in,// 8-bit inputs
  input signed [W1-1:0]  cps_in, cms_in,  // coefficients
  output reg signed [W-1:0]  Dre_out, Dim_out,// registered
  output signed   [W-1:0]  Ere_out, Eim_out);  // results

  reg signed [W-1:0] dif_re, dif_im;      // Bf out
```

```
  reg signed [W-1:0] Are, Aim, Bre, Bim;  // Inputs integer
  reg signed [W-1:0] c;                   // Input
  reg signed [W1-1:0] cps, cms;           // Coefficient in

  always @(posedge clk)   // Compute the additions of the
  begin                   // butterfly using integers
    Are     <= Are_in;    // and store inputs
    Aim     <= Aim_in;    // in flip-flops
    Bre     <= Bre_in;
    Bim     <= Bim_in;
    c       <= c_in;            // Load from memory cos
    cps     <= cps_in;          // Load from memory cos+sin
    cms     <= cms_in;          // Load from memory cos-sin
    Dre_out <= (Are >>> 1) + (Bre >>> 1); // Are/2 + Bre/2
    Dim_out <= (Aim >>> 1) + (Bim >>> 1); // Aim/2 + Bim/2
  end

     // No FF because butterfly difference "diff" is not an
  always @(*)                               // output port
  begin
    dif_re = (Are >>> 1) - (Bre >>> 1);//i.e. Are/2 - Bre/2
    dif_im = (Aim >>> 1) - (Bim >>> 1);//i.e. Aim/2 - Bim/2
  end

  //*** Instantiate the complex twiddle factor multiplier
  ccmul ccmul_1                    // Multiply (x+jy)(c+js)
  ( .clk(clk), .x_in(dif_re), .y_in(dif_im),  .c_in(c),
    .cps_in(cps), .cms_in(cms), .r_out(Ere_out),
                                          .i_out(Eim_out));

endmodule

//*********************************************************
// IEEE STD 1364-2001 Verilog file: lfsr.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*********************************************************
module lfsr           //----> Interface
  (input      clk,
  output [6:1]  y);  // Result

  reg [6:1] ff; // Note that reg is keyword in Verilog and
                              // can not be variable name
  integer i;

  always @(posedge clk) begin // Length-6 LFSR with xnor
```

```
    ff[1] <= ff[5] ~^ ff[6]; // Use nonblocking assignment
    for (i=6; i>=2 ; i=i-1) // Tapped delay line: shift one
      ff[i] <= ff[i-1];
  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,
  output [6:1]  y);  // Result

  reg [6:1] ff; // Note that reg is keyword in Verilog and
                              // can not be variable name

  always @(posedge clk) begin // Implement three-step
    ff[6] <= ff[3];           // length-6 LFSR with xnor;
    ff[5] <= ff[2];         // use nonblocking assignments
    ff[4] <= ff[1];
    ff[3] <= ff[5] ~^ ff[6];
    ff[2] <= ff[4] ~^ ff[5];
    ff[1] <= ff[3] ~^ ff[4];
  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,              //----> Interface
  input signed [W:0] r_in,
  input signed [W:0] phi_in,
  output reg signed [W:0] x_out, y_out, eps);

  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) begin //----> Infer register
  if (phi_in > 90)           // Test for |phi_in| > 90
    begin                    // Rotate 90 degrees
    x[0] <= 0;
    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)                  // Rotate 45 degrees
    begin
    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)                // 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] - '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)                    // 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] - 'sd14;
      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] + 'sd14;
      end

    x_out <= x[3];
    eps   <= z[3];
    y_out <= y[3];
  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,  // 1 bit input
  input signed [W1-1:0] x_in, d_in,  // Inputs
  output signed [W2-1:0] e_out, y_out,  // Results
  output signed [W1-1:0] f0_out, f1_out);  // Results

// 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;
  wire signed [W2-1:0] p [0:1]; // 1. Product array
  wire signed [W2-1:0] xemu [0:1]; // 2. Product array
  wire signed [W2-1:0]  y, sxty, e, sxtd;

  wire  clken, aclr;
  wire  signed [W2-1:0] sum;  // Auxilary signals


  assign sum=0; assign aclr=0; // Default for mult
  assign clken=0;

  always @(posedge clk) // Store these data or coefficients
    begin: Store
      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

// Instantiate L pipelined multiplier
  genvar I;
  generate
    for (I=0; I<L; I=I+1) begin: Mul_fx
  lpm_mult mul_xf             // Multiply  x[I]*f[I] = p[I]
    ( .dataa(x[I]), .datab(f[I]), .result(p[I]));
//   .clock(clk), .sum(sum),
//   .clken(clken), .aclr(aclr)); // Unused ports
    defparam mul_xf.lpm_widtha = W1;
    defparam mul_xf.lpm_widthb = W1;
    defparam mul_xf.lpm_widthp = W2;
    defparam mul_xf.lpm_widths = W2;
//    defparam mul_xf.lpm_pipeline = Delay;
    defparam mul_xf.lpm_representation = "SIGNED";
    end // for loop
  endgenerate



  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 pipelined multiplier
  generate
    for (I=0; I<L; I=I+1) begin: Mul_xemu
  lpm_mult mul_I          // Multiply xemu[I] = emu * x[I];
    ( .dataa(x[I]), .datab(emu), .result(xemu[I]));
//   .clock(clk), .sum(sum),
//   .clken(clken), .aclr(aclr)); // Unused ports
    defparam mul_I.lpm_widtha = W1;
    defparam mul_I.lpm_widthb = W1;
    defparam mul_I.lpm_widthp = W2;
    defparam mul_I.lpm_widths = W2;
//    defparam mul_I.lpm_pipeline = Delay;
    defparam mul_I.lpm_representation = "SIGNED";
    end // for loop
  endgenerate

  assign  y_out  = y;    // 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: fir6dlms.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//************************************************************
// This is a generic DLMS FIR filter generator
// It uses W1 bit data/coefficients bits

module fir6dlms        //----> 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,  // 1 bit input
  input signed [W1-1:0] x_in, d_in,  // Inputs
  output signed [W2-1:0] e_out, y_out,  // Results
  output signed [W1-1:0] f0_out, f1_out);  // Results
```

```
// 2D array types memories are supported by Quartus II
// in Verilog, use therefore single vectors
  reg signed [W1-1:0] x [0:4], f0, f1;
  reg signed [W1-1:0] f[0:1];
  reg  signed [W1-1:0] d[0:3]; // Desired signal array
  wire signed [W1-1:0] emu;
  wire signed [W2-1:0] xemu[0:1]; // Product array
  wire signed [W2-1:0] p[0:1]; // Product array
  wire  signed [W2-1:0]  y, sxty, e, sxtd;

  wire  clken, aclr;
  wire  signed [W2-1:0] sum;  // Auxilary signals


  assign sum=0; assign aclr=0; // Default for mult
  assign clken=0;

  always @(posedge clk) // Store these data or coefficients
    begin: Store
      d[0] <= d_in; // Shift register for desired data
      d[1] <= d[0];
      d[2] <= d[1];
      d[3] <= d[2];
      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

// Instantiate L pipelined multiplier
  genvar I;
  generate
    for (I=0; I<L; I=I+1) begin: Mul_fx
  lpm_mult mul_xf             // Multiply  x[I]*f[I] = p[I]
  (.clock(clk), .dataa(x[I]), .datab(f[I]), .result(p[I]));
//  .sum(sum), .clken(clken), .aclr(aclr)); // Unused ports
    defparam mul_xf.lpm_widtha = W1;
    defparam mul_xf.lpm_widthb = W1;
    defparam mul_xf.lpm_widthp = W2;
    defparam mul_xf.lpm_widths = W2;
```

```
    defparam mul_xf.lpm_pipeline = Delay;
    defparam mul_xf.lpm_representation = "SIGNED";
    end // for loop
  endgenerate

  assign y = p[0] + p[1];  // Compute ADF output

  // Scale y by 128 because x is fraction
  assign e = d[3] - (y >>> 7);
  assign emu = e >>> 1;  // e*mu divide by 2 and
                         // 2 from xemu makes mu=1/4

// Instantiate L pipelined multiplier
  generate
    for (I=0; I<L; I=I+1) begin: Mul_xemu
  lpm_mult mul_I          // Multiply xemu[I] = emu * x[I];
    (.clock(clk), .dataa(x[I+Delay]), .datab(emu),
                                        .result(xemu[I]));
//  .sum(sum), .clken(clken), .aclr(aclr)); // Unused ports
    defparam mul_I.lpm_widtha = W1;
    defparam mul_I.lpm_widthb = W1;
    defparam mul_I.lpm_widthp = W2;
    defparam mul_I.lpm_widths = W2;
    defparam mul_I.lpm_pipeline = Delay;
    defparam mul_I.lpm_representation = "SIGNED";
    end // for loop
  endgenerate

  assign  y_out  = y;    // Monitor some test signals
  assign  e_out  = e;
  assign  f0_out = f[0];
  assign  f1_out = f[1];

endmodule

// Desciption: This is a W x L bit register file.
//*************************************************************
// IEEE STD 1364-2001 Verilog file: reg_file.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*************************************************************
module reg_file  #(parameter W = 7, // Bit width -1
                            N  = 15) //Number of register - 1
       (input clk, reg_ena,
        input [W:0] data,
        input [3:0]  rd, rs, rt ,
```

```
        output reg [W:0] s, t);

  reg [W:0] r [0:N];

  always @(posedge clk) // Input mux inferring registers
  begin : MUX
    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 three-phase clock cycle design
// It has a stack machine/0-address-type instruction word
// The stack has only four words.
//`include "220model.v"

module trisc0 #(parameter WA = 7,  // Address bit width -1
                          WD = 7)    // Data bit width -1
 (input reset, clk,  // Clock for the output register
  output  jc_OUT, me_ena,
  input [WD:0] iport,
  output reg [WD:0] oport,
  output [WD:0] s0_OUT, s1_OUT, dmd_IN, dmd_OUT,
  output [WA:0] pc_OUT, dma_OUT, dma_IN,
  output [7:0]  ir_imm,
```

```
  output [3:0]  op_code);

  //parameter ifetch=0, load=1, store=2, incpc=3;
  reg [1:0] state;

  wire [3:0] op;
  wire [WD:0] imm, dmd;
  reg [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;

// Code of FSM:
  always @(op) // 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

  lpm_rom prog_rom
  ( .outclock(clk),.address(pc), .q(pmd));  // Used ports
// .inclock(clk),  .memenab(ena)); // Unused
    defparam prog_rom.lpm_width = 12;
    defparam prog_rom.lpm_widthad = 8;
    defparam prog_rom.lpm_outdata = "REGISTERED";
    defparam prog_rom.lpm_address_control = "UNREGISTERED";
    defparam prog_rom.lpm_file = "TRISC0FAC.MIF";

  assign not_clk = ~clk;

  lpm_ram_dq data_ram
  ( .inclock(not_clk),.address(dma), .q(dmd),
    .data(s0), .we(mem_ena));  // Used ports
// .outclock(clk)); // Unused
    defparam data_ram.lpm_width = 8;
    defparam data_ram.lpm_widthad = 8;
    defparam data_ram.lpm_indata = "REGISTERED";
    defparam data_ram.lpm_outdata = "UNREGISTERED";
    defparam data_ram.lpm_address_control = "REGISTERED";


  always @(posedge clk)
  begin : P3
    integer temp;

    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

  // 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 = pmd; 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
```

B. VHDL and Verilog Coding

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 1076-1987 uses 77 keywords and an extra 19 keywords are used in VHDL 1076-1993 (see VHDL 1076-1993 Language Reference Manual (LRM) on p. 179). New in VHDL 1076-1993 are:

```
GROUP, IMPURE, INERTIAL, LITERAL, POSTPONED, PURE, REJECT
 ROL, ROR, SHARED, SLA, SLL, SRA, SRL, UNAFFECTED, XNOR,
```

which are unfortunately *not* highlighted in the MaxPlus II editor but with the Quartus II. The IEEE standard Verilog 1364-1995, on the other hand, has 102 keywords (see LRM, p. 604). Together, both HDL languages have 201 keywords, including 19 in common. Table B.1 shows VHDL 1076-1993 keywords in capital letters, Verilog 1364-2001 keywords in small letters, and the common keywords with a capital first letter. New in Verilog 1076-2001 are:

```
automatic, cell, config, design, endconfig, endgenerate,
generate, genvar, incdir, include, instance, liblist,
library, localparam, noshowcancelled, pulsestyle_onevent,
pulsestyle_ondetect, showcancelled, signed, unsigned, use
```

Table B.1. VHDL 1076-1993 and Verilog 1364-2001 keywords.

ABS	event	notif0	
ACCESS	EXIT	notif1	SIGNAL
AFTER	FILE	NULL	signed
ALIAS	For	OF	OF
ALL	force	ON	SLA
always	forever	OPEN	SLL
And	fork	Or	small
ARCHITECTURE	Function	OTHERS	specify
ARRAY	Generate	OUT	specparam
ASSERT	GENERIC	output	SRA
assign	genvar	PACKAGE	SRL
ATTRIBUTE	GROUP	parameter	strong0
automatic	GUARDED	pmos	strong1
Begin	highz0	PORT	SUBTYPE
BLOCK	highz1	posedge	supply0
BODY	If	POSTPONED	supply1
buf	ifnone	primitive	table
BUFFER	IMPURE	PROCEDURE	task
bufif0	IN	PROCESS	THEN
bufif1	incdir	pull0	time
BUS	include	pull1	TO
Case	INERTIAL	pulldown	tran
casex	initial	pullup	tranif0
casez	Inout	pulsestyle_onevent	tranif1
cell	input	pulsestyle_ondetect	TRANSPORT
cmos	instance	PURE	tri
config	integer	RANGE	tri0
COMPONENT	IS	rcmos	tri1
CONFIGURATION	join	real	triand
CONSTANT	LABEL	realtime	trior
deassign	large	RECORD	trireg
default	liblist	reg	TYPE
defparam	Library	REGISTER	UNAFFECTED
design	LINKAGE	REJECT	UNITS
disable	LITERAL	release	unsigned
DISCONNECT	LOOP	REM	UNTIL
DOWNTO	localparam	repeat	Use
edge	macromodule	REPORT	VARIABLE
Else	MAP	RETURN	vectored
ELSIF	medium	rnmos	Wait
End	MOD	ROL	wand
endcase	module	ROR	weak0
endconfig	Nand	rpmos	weak1
endfunction	negedge	rtran	WHEN
endgenerate	NEW	rtranif0	While
endmodule	NEXT	rtranif1	wire
endprimitive	nmos	scalared	WITH
endspecify	Nor	SELECT	wor
endtable	noshowcancelled	SEVERITY	Xnor
endtask	Not	SHARED	Xor
ENTITY		showcancelled	

B.1 List of Examples

These synthesis results for all examples can be easily reproduced by using the scripts `qvhdl.tcl` in the VHDL or Verilog directories of the CD-ROM. Run the TCL script with

```
quartus_sh -t qvhdl.tcl > qvhdl.txt
```

The script produces for each design four parameters. For the `trisc0.vhd`, for instance, we get:

```
....
-----------------------------------------------------
trisc0 fmax: 115.65 MHz ( period = 8.647 ns )
trisc0 LEs: 198 / 33,216 ( < 1 % )
trisc0 M4K bits: 5,120 / 483,840 ( 1 % )
trisc0 DSP blocks: 1 / 70 ( 1 % )
-----------------------------------------------------
....
```

then `grep` through the report `qvhdl.txt` file using `fmax:`, `LEs:` etc.

From the script you will notice that the following special options of Quartus II web edition 6.0 were used:

- `Device` set `Family` to `Cyclone II` and then under `Available devices` select `EP2C35F672C6`.
- For `Timing Analysis Settings` set `Default required fmax:` to 3 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 M4K memory blocks; the `Registered Performance`; 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 number of 9×9-bit multiplier and number of M4K memory blocks, but the LEs and `Registered Performance` do not always match.

Design	VHDL LEs	VHDL 9×9 Mult.	VHDL M4Ks	VHDL f_{MAX} MHz	VHDL Page	Verilog LEs	Verilog f_{MAX} MHz	Verilog Page
add_1p	125	no	0	316.46	78	77	390.63	666
add_2p	234	no	0	229.04	78	144	283.85	667
add_3p	372	no	0	215.84	78	229	270.42	668
ammod	316	no	0	215.98	455	277	288.85	717
arctan	100	4	0	32.09	134	99	32.45	676
bfproc	131	3	0	95.73	370	83	116.09	715
ccmul	39	3	0	–	368	39	–	713
cic3r32	337	no	0	282.17	262	337	269.69	694
cic3s32	205	no	0	284.58	269	205	284.50	696
cmoms	372	10	0	85.94	303	239	107.48	705
cmul7p8	48	no	0	-	59	48	–	665
cordic	235	no	0	222.67	126	197	317.16	674
dafsm	32	no	0	420.17	189	30	420.17	683
dapara	33	no	0	214.96	202	45	420.17	691
darom	27	no	1	218.29	196	27	218.96	687
dasign	56	no	0	236.91	199	47	328.19	688
db4latti	418	no	0	58.81	324	248	74.69	709
db4poly	173	no	0	136.65	250	158	136.31	697
div_aegp	64	4	0	134.63	94	64	134.63	671
div_res	127	no	0	265.32	100	115	257.86	673
example	24	no	0	420.17	15	24	420.17	663
farrow	279	6	0	43.91	292	175	65.77	703
fir6dlms	138	4	0	176.15	511	138	174.52	721
fir_gen	184	4	0	329.06	167	184	329.06	680
fir_lms	50	4	0	74.59	504	50	74.03	719
fir_srg	114	no	0	97.21	179	70	106.15	682
fun_text	32	no	1	264.20	30	32	264.20	664
iir	62	no	0	160.69	217	30	234.85	692
iir_par	268	no	0	168.12	237	199	136.87	693
iir_pipe	124	no	0	207.08	231	75	354.48	692
lfsr	6	no	0	420.17	437	6	420.17	716
lfsr6s3	6	no	0	420.17	440	6	420.17	717
ln	88	10	0	32.76	145	88	32.76	677
mul_ser	121	no	0	256.15	82	140	245.34	670
rader7	443	no	0	137.06	355	404	159.41	710
rc_sinc	448	19	0	61.93	285	416	81.47	699
reg_file	211	no	0	-	559	211	–	723
sqrt	336	2	0	82.16	150	317	82.73	678
tris0	198	1	2	115.65	606	166	71.94	724

B.2 Library of Parameterized Modules (LPM)

Throughout the book we use six different LPM megafunctions (see Fig. B.1), namely:

- `lpm_ff`, the flip-flop megafunction
- `lpm_add_sub`, the adder/subtractor megafunction
- `lpm_ram_dq`, the RAM megafunction
- `lpm_rom`, the ROM megafunction
- `lpm_divide`, the divider megafunction, and
- `lpm_mult`, the multiplier megafunction

These megafunctions are explained in the following, along with their port definitions, parameters, and resource usage. This information is also available using the Quartus II help under `megafunctions/LPM`.

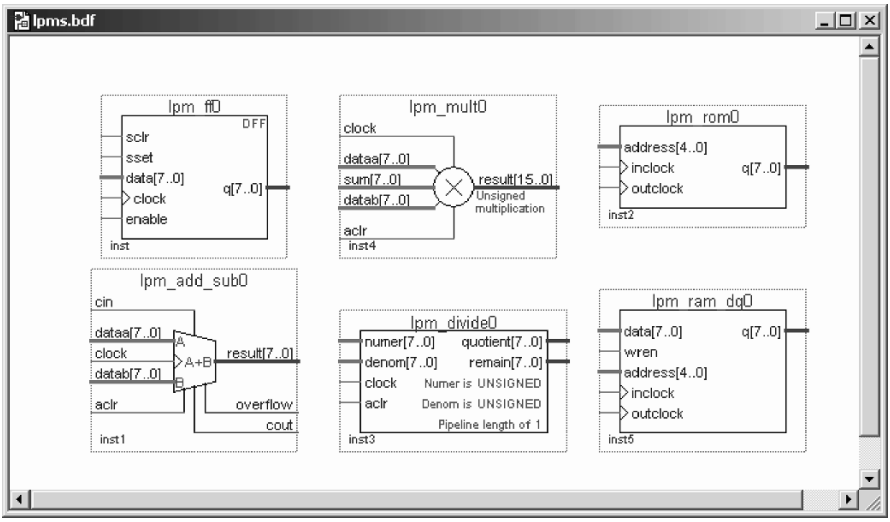

Fig. B.1. Six LPM megafunction used.

B.2.1 The Parameterized Flip-Flop Megafunction (lpm_ff)

The `lpm_ff` function is useful if features are needed that are not available in the `DFF`, `DFFE`, `TFF`, and `TFFE` primitives, such as synchronous or asynchronous set, clear, and load inputs. We have used this megafunction for the following designs: `example`, p. 15 and `fun_text`, p. 30.

Altera recommends instantiating this function as described in "Using the MegaWizard Plug-In Manager" in the Quartus II help.
The port names and order for Verilog HDL prototypes are:

```
module lpm_ff ( q, data, clock, enable, aclr,
                aset, sclr, sset, aload, sload) ;
```

The VHDL component declaration is shown below:

```
COMPONENT lpm_ff
   GENERIC (LPM_WIDTH: POSITIVE;
            LPM_AVALUE: STRING := "UNUSED";
            LPM_FFTYPE: STRING := "FFTYPE_DFF";
            LPM_TYPE: STRING := "L_FF";
            LPM_SVALUE: STRING := "UNUSED";
            LPM_HINT: STRING := "UNUSED");
   PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
        clock: IN STD_LOGIC;
       enable: IN STD_LOGIC := '1';
        sload: IN STD_LOGIC := '0';
         sclr: IN STD_LOGIC := '0';
         sset: IN STD_LOGIC := '0';
        aload: IN STD_LOGIC := '0';
         aclr: IN STD_LOGIC := '0';
         aset: IN STD_LOGIC := '0';
            q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;
```

Ports

The following table displays all input ports of `lpm_ff`:

Port name	Required	Description	Comments
data	No	T-type flip-flop: Toggle enable D-type flip-flop: Data input	Input port LPM_WIDTH wide. If the data input is not used, at least one of the aset, aclr, sset, or sclr ports must be used. Unused data inputs default to GND.
clock	Yes	Positive-edge triggered clock	
enable	No	Clock Enable input	Default = 1
sclr	No	Synchronous clear input	If both sset and sclr are used and both are asserted, sclr is dominant. The sclr signal affects the output q values before polarity is applied to the ports.
sset	No	Synchronous set input	Sets q outputs to the value specified by LPM_SVALUE, if that value is present, or sets the q outputs to all 1s. If both sset and sclr are used and both are asserted, sclr is dominant. The sset signal affects the output q values before polarity is applied to the ports.
sload	No	Synchronous load input. Loads the flip-flop with the value on the data input on the next active clock edge.	Default = 0. If sload is used, data must be used. For load operation, sload must be high (1) and enable must be high (1) or unconnected. The sload port is ignored when the LPM_FFTYPE parameter is set to DFF.
aclr	No	Asynchronous clear input	If both aset and aclr are used and both are asserted, aclr is dominant. The aclr signal affects the output q values before polarity is applied to the ports.
aset	No	Asynchronous set input	Sets q outputs to the value specified by LPM_AVALUE, if that value is present, or sets the q outputs to all 1s.
aload	No	Asynchronous load input. Asynchronously loads the flip-flop with the value on the data input.	Default = 0. If aload is used, data must be used.

The following table displays all OUTPUT ports of lpm_ff:

Port Name	Required	Description	Comments
q	Yes	Data output from D or T flip-flops	Output port LPM_WIDTH wide

Parameters

The following table shows the parameters of the `lpm_ff` component:

Parameter	Type	Required	Description
`LPM_WIDTH`	Integer	Yes	Width of the `data` and `q` ports
`LPM_AVALUE`	Integer	No	Constant value that is loaded when `aset` is high. If omitted, defaults to all 1s. The `LPM_AVALUE` parameter is limited to a maximum of 32 bits.
`LPM_SVALUE`	Integer	No	Constant value that is loaded on the rising edge of `clock` when `sset` is high. If omitted, defaults to all 1s.
`LPM_FFTYPE`	String	No	Values are `DFF`, `TFF`, and `UNUSED`. Type of flip-flop. If omitted, the default is `DFF`. When the `LPM_FFTYPE` parameter is set to `DFF`, the `sload` port is ignored.
`LPM_HINT`	String	No	Allows you to specify Altera-specific parameters in VHDL design files. The default is `UNUSED`.
`LPM_TYPE`	String	No	Identifies the LPM entity name in the VHDL design files.

Note that for Verilog LPM 220 synthesizable code (i.e., `220model.v`) the following parameter ordering applies: `lpm_type`, `lpm_width`, `lpm_avalue`, `lpm_svalue`, `lpm_pvalue`, `lpm_fftype`, `lpm_hint`.

Function

The following table is an example of the T-type flip-flop behavior in `lpm_ff`:

Inputs							Outputs
aclr	aset	enable	clock	sclr	sset	sload	`Q[LPM_WIDTH-1..0]`
1	X	X	X	X	X	X	000...
0	1	X	X	X	X	X	111... or `LPM_AVALUE`
0	0	0	X	X	X	X	`q[LPM_WIDTH-1..0]`
0	0	1	⨍	1	X	X	000...
0	0	1	⨍	0	1	X	111... or `LPM_SVALUE`
0	0	1	⨍	0	0	1	`data[LPM_WIDTH-1..0]`
0	0	1	⨍	0	0	0	`q[LPM_WIDTH-1..0]` xor `data[LPM_WIDTH-1..0]`

Resource Usage

The megafunction `lpm_ff` uses one logic cell per bit.

B.2.2 The Parameterized Adder/Subtractor Megafunction (lpm_add_sub)

Altera recommends using the `lpm_add_sub` function to replace all other types of adder/subtractor functions, including old-style adder/subtractor macrofunctions. We have used this megafunction for the following designs: `example`, p. 15, `fun_text`, p. 30, `ccmul`, p. 368.

Altera recommends instantiating this function as described in "Using the MegaWizard Plug-In Manager" in the Quartus II help.
The port names and order for Verilog HDL prototypes are:

```
module lpm_add_sub ( cin,
                     dataa, datab,
                     add_sub, clock, aclr,
                     result, cout, overflow);
```

The VHDL component declaration is shown below:

```
COMPONENT lpm_add_sub
  GENERIC (LPM_WIDTH: POSITIVE;
           LPM_REPRESENTATION: STRING := "SIGNED";
           LPM_DIRECTION: STRING := "UNUSED";
           LPM_HINT: STRING := "UNUSED";
           LPM_PIPELINE: INTEGER := 0;
           LPM_TYPE: STRING := "L_ADD_SUB");
  PORT (dataa, datab
              : IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
        aclr, clken, clock, cin : IN STD_LOGIC := '0';
        add_sub                 : IN STD_LOGIC := '1';
        result : OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
        cout, overflow          : OUT STD_LOGIC);
END COMPONENT;
```

Ports

The following table displays all input ports of `lpm_add_sub`:

Port name	Required	Description	Comments
`cin`	No	Carry-in to the low-order bit. If the operation is `ADD`, low = 0 and high = +1. If the operation is `SUB`, low = −1 and high = 0.	If omitted, the default is 0 (i.e., low if the operation is `ADD` and high if the operation is `SUB`).
`dataa`	Yes	Augend/Minuend	Input port `LPM_WIDTH` wide
`datab`	Yes	Addend/Subtrahend	Input port `LPM_WIDTH` wide
`add_sub`	No	If the signal is high, the operation = `dataa` + `datab`. If the signal is low, the operation = `dataa` − `datab`.	If the `LPM_DIRECTION` parameter is used, `add_sub` cannot be used. If omitted, the default is `ADD`. Altera recommends that you use the `LPM_DIRECTION` parameter to specify the operation of the `lpm_add_sub` function, rather than assigning a constant to the `add_sub` port.
`clock`	No	Clock for pipelined usage	The clock port provides pipelined operation for the `lpm_add_sub` function. For `LPM_PIPELINE` values other than 0 (default value), the `clock` port must be connected.
`clken`	No	Clock enable for pipelined usage	Available for VHDL only
`aclr`	No	Asynchronous clear for pipelined usage	The pipeline initializes to an undefined (X) logic level. The `aclr` port can be used at any time to reset the pipeline to all 0s, asynchronously to the `clock` signal.

The following table displays all output ports of `lpm_add_sub`:

Port Name	Required	Description	Comments
`result`	Yes	`dataa` + or − `datab` + or − `cin`	Output port `LPM_WIDTH` wide
`cout`	No	Carry-out (borrow-in) of the MSB	If `overflow` is used, `cout` cannot be used. The `cout` port has a physical interpretation as the carry-out (borrow-in) of the MSB. `cout` is most meaningful for detecting overflow in `UNSIGNED` operations.
`overflow`	No	Result exceeds available precision.	If `overflow` is used, `cout` cannot be used. The `overflow` port has a physical interpretation as the `XOR` of the carry-in to the MSB with the carry-out of the MSB. `overflow` is meaningful only when the `LPM_REPRESENTATION` parameter value is `SIGNED`.

Parameters

The following table shows the parameters of the `lpm_add_sub` component:

Parameter	Type	Required	Description
LPM_WIDTH	Integer	Yes	Width of the dataa, datab, and result ports.
LPM_DIRECTION	String	No	Values are ADD, SUB, and UNUSED. If omitted, the default is DEFAULT, which directs the parameter to take its value from the add_sub port. The add_sub port cannot be used if LPM_DIRECTION is used. Altera recommends that you use the LPM_DIRECTION parameter to specify the operation of the lpm_add_sub function, rather than assigning a constant to the add_sub port.
LPM_REPRESENTATION	String	No	Type of addition performed: SIGNED, UNSIGNED, or UNUSED. If omitted, the default is SIGNED.
LPM_PIPELINE	Integer	No	Specifies the number of clock cycles of latency associated with the result output. A value of zero (0) indicates that no latency exists, and that a purely combinatorial function will be instantiated. If omitted, the default is 0 (nonpipelined).
LPM_HINT	String	No	Allows you to specify Altera-specific parameters in VHDL design files. The default is UNUSED.
LPM_TYPE	String	No	Identifies the LPM entity name in VHDL design files.
ONE_INPUT_IS_CONSTANT	String	No	Altera-specific parameter. Values are YES, NO, and UNUSED. Provides greater optimization, if one input is constant. If omitted, the default is NO.
MAXIMIZE_SPEED	Integer	No	Altera-specific parameter. You can specify a value between 0 and 10. If used, MaxPlus II attempts to optimize a specific instance of the lpm_add_sub function for speed rather than area, and overrides the setting of the Optimize option in the Global Project Logic Synthesis dialog box (Assign menu). If MAXIMIZE_SPEED is unused, the value of the Optimize option is used instead. If the setting for MAXIMIZE_SPEED is 6 or higher, the compiler will optimize lpm_add_sub megafunctions for higher speed; if the setting is 5 or less, the compiler will optimize for smaller area.

Note that for Verilog LPM 220 synthesizable code (i.e., 220model.v) the following parameter ordering applies: lpm_type, lpm_width, lpm_direction, lpm_representation, lpm_pipeline, lpm_hint.

Function

The following table is an example of the UNSIGNED behavior in `lpm_add_sub`:

Inputs			Outputs	
`add_sub`	`dataa`	`datab`	`cout,result`	`overflow`
1	a	b	`a + b + cin`	`cout`
0	a	b	`a - b - cin`	`!cout`

The following table is an example of the SIGNED behavior in `lpm_add_sub`:

Inputs			Outputs	
`add_sub`	`dataa`	`datab`	`cout,sum`	`overflow`
1	a	b	$a + b$+`cin`	$a \geq 0$ and $b \geq 0$ and $sum < 0$ or $a < 0$ and $b < 0$ and $sum \geq 0$
0	a	b	$a - b$−`cin`	$a >= 0$ and $b < 0$ and $sum < 0$ or $a < 0$ and $b \geq 0$ and $sum \geq 0$

Resource Usage

The following table summarizes the resource usage for an `lpm_add_sub` megafunction used to implement a 16-bit unsigned adder with a carry-in input and a carry-out output. Logic cell usage scales linearly in proportion to adder width.

Design goals		Design results		
Device family	Optimization	LEs	Speed (ns)	Notes
FLEX 6K, 8K, and 10K	Routability	45	53	Speed for EPF8282A-2
	Speed	18	17	
MAX 5K, 7K, and 9K	Routability	28 (22)	23	Speed for EPM7128E-7

Numbers of shared expanders used are shown in parentheses.

B.2.3 The Parameterized Multiplier Megafunction (lpm_mult)

Altera recommends that you use `lpm_mult` to replace all other types of multiplier functions, including old-style multiplier macrofunctions. We have used

this megafunction for the designs `fir_gen`, p. 167, `ccmul`, p. 368. `fir_lms`, p. 504, and `fir6dlms`, p. 511.

Altera recommends instantiating this function as described in "Using the MegaWizard Plug-In Manager" in the Quartus II help.

The port names and order for Verilog HDL prototype are:

```
module lpm_mult ( dataa, datab, sum, aclr, clock,
                  result);
```

The VHDL component declaration is shown below:

```
COMPONENT lpm_mult
  GENERIC (LPM_WIDTHA: POSITIVE;
           LPM_WIDTHB: POSITIVE;
           LPM_WIDTHS: POSITIVE;
           LPM_WIDTHP: POSITIVE;
           LPM_REPRESENTATION: STRING := "UNSIGNED";
           LPM_PIPELINE: INTEGER := 0;
           LPM_TYPE: STRING := "L_MULT";
           LPM_HINT : STRING := "UNUSED");
  PORT (dataa : IN STD_LOGIC_VECTOR(LPM_WIDTHA-1 DOWNTO 0);
        datab : IN STD_LOGIC_VECTOR(LPM_WIDTHB-1 DOWNTO 0);
        aclr, clken, clock : IN STD_LOGIC := '0';
        sum   : IN STD_LOGIC_VECTOR(LPM_WIDTHS-1 DOWNTO 0)
                                          := (OTHERS => '0');
        result: OUT STD_LOGIC_VECTOR(LPM_WIDTHP-1 DOWNTO 0)
        );
END COMPONENT;
```

Ports

The following table displays all input ports of `lpm_mult`:

Port name	Required	Description	Comments
`dataa`	Yes	Multiplicand	Input port `LPM_WIDTHA` wide
`datab`	Yes	Multiplier	Input port `LPM_WIDTHB` wide
`sum`	No	Partial sum	Input port `LPM_WIDTHS` wide
`clock`	No	Clock for pipelined usage	The `clock` port provides pipelined operation for the `lpm_mult` function. For `LPM_PIPELINE` values other than 0 (default value), the `clock` port must be connected.
`clken`	No	Clock enable for pipelined usage	Available for VHDL only.
`aclr`	No	Asynchronous clear for pipelined usage	The pipeline initializes to an undefined (X) logic level. The `aclr` port can be used at any time to reset the pipeline to all 0s, asynchronously to the `clock` signal.

The following table displays all output ports of `lpm_mult`:

Port Name	Required	Description	Comments
`result`	Yes	result = `dataa * datab + sum`. The product LSB is aligned with the sum LSB.	Output port `LPM_WIDTHP` wide. If `LPM_WIDTHP` < max `(LPM_WIDTHA + LPM_WIDTHB, LPM_WIDTHS)` or `(LPM_WIDTHA + LPM_WIDTHS)`, only the `LPM_WIDTHP` MSBs are present.

Parameters

The following table shows the parameters of the `lpm_mult` component:

Parameter	Type	Required	Description
LPM_WIDTHA	Integer	Yes	Width of the dataa port
LPM_WIDTHB	Integer	Yes	Width of the datab port
LPM_WIDTHP	Integer	Yes	Width of the result port
LPM_WIDTHS	Integer	Yes	Width of the sum port. Required even if the sum port is not used.
LPM_REPRESENTATION	String	No	Type of multiplication performed: SIGNED, UNSIGNED, or UNUSED. If omitted, the default is UNSIGNED.
LPM_PIPELINE	Integer	No	Specifies the number of clock cycles of latency associated with the result output. A value of zero (0) indicates that no latency exists, and that a purely combinatorial function will be instantiated. If omitted, the default is 0 (non-pipelined).
LPM_HINT	String	No	Allows you to assign Altera-specific parameters in VHDL design files. The default is UNUSED.
LPM_TYPE	String	No	Identifies the LPM entity name in VHDL design files.
INPUT_A_IS_CONSTANT	String	No	Altera-specific parameter. Values are YES, NO, and UNUSED. If dataa is connected to a constant value, setting INPUT_A_IS_CONSTANT to YES optimizes the multiplier for resource usage and speed. If omitted, the default is NO.
INPUT_B_IS_CONSTANT	String	No	Altera-specific parameter. Values are YES, NO, and UNUSED. If datab is connected to a constant value, setting INPUT_B_IS_CONSTANT to YES optimizes the multiplier for resource usage and speed. The default is NO.

Parameter	Type	Required	Description
USE_EAB	String	No	Altera-specific parameter. Values are ON, OFF, and UNUSED. Setting the USE_EAB parameter to ON allows Quartus II to use EABs to implement 4 × 4 or (8 × constant value) building blocks in FLEX 10K devices. Altera recommends that you set USE_EAB to ON only when LCELLS are in short supply. If you wish to use this parameter, when you instantiate the function in a GDF, you must specify it by entering the parameter name and value manually with the Edit Ports/Parameters dialog box (Symbol menu). You can also use this parameter name in a TDF or a Verilog design file. You must use the LPM_HINT parameter to specify the USE_EAB parameter in VHDL design files.
DEDICATED_ _MULTIPLIER_ _CIRCUITRY	String	No	Altera-specific parameter. You must use the LPM_HINT parameter to specify the DEDICATED_MULTIPLIER_CIRCUITRY parameter in VHDL design files. Specifies whether to use dedicated multiplier circuitry. Values are 'AUTO, YES, and 'NO,. If omitted, the default is AUTO.
LATENCY	Integer	No	Altera-specific parameter. Same as LPM_PIPELINE. (This parameter is provided only for backward compatibility with MaxPlus II pre-version 7.0 designs. For all new designs, you should use the LPM_PIPELINE parameter instead.)
MAXIMIZE_ _SPEED	Integer	No	Altera-specific parameter. You can specify a value between 0 and 10. If used, MaxPlus II attempts to optimize a specific instance of the lpm_mult function for speed rather than area, and overrides the setting of the Optimize option in the Global Project Logic Synthesis dialog box (Assign menu). If MAXIMIZE_SPEED is unused, the value of the Optimize option is used instead. If the setting for MAXIMIZE_SPEED is 6 or higher, the compiler will optimize lpm_mult megafunctions for higher speed; if the setting is 5 or less, the compiler will optimize for smaller area.
LPM_HINT	String	No	Allows you to specify Altera-specific parameters in VHDL design files. The default is UNUSED.

Note that specifying a value for MAXIMIZE_SPEED has an effect only if LPM_REPRESENTATION is set to SIGNED.

Note that for Verilog LPM 220 synthesizable code (i.e., 220model.v) the following parameter ordering applies: lpm_type, lpm_widtha, lpm_widthb, lpm_widths, lpm_widthp, lpm_representation, lpm_pipeline, lpm_hint.

Function

The following table is an example of the `UNSIGNED` behavior in `lpm_mult`:

Inputs			Outputs
`dataa`	`datab`	`sum`	`product`
a	b	s	`LPM_WIDTHP` most significant bits of $a * b + s$

Resource Usage

The following table summarizes the resource usage for an `lpm_mult` function used to implement 4-bit and 8-bit multipliers with `LPM_PIPELINE` = 0 and without the optional `sum` input. Logic cell usage scales linearly in proportion to the square of the input width.

Design goals		Design results			
Device family	Optimization	Width	LEs	Speed (ns)	Notes
FLEX 6K, 8K, and 10K	Routability	8	121	80	Speed for
	Speed	8	163	52	EPF8282A-2
FLEX 6K, 8K, and 10K	Routability	4	29	34	Speed for
	Speed	4	41	27	EPF8282A-2
MAX 5K, 7K, and 9K	Routability	4	26 (11)	23	Speed for
	Speed	4	27 (4)	19	EPM7128E-7

Numbers of shared expanders used are shown in parentheses. In the FLEX 10K device family, the 4-bit by 4-bit multiplier example shown above can be implemented in a single EAB.

B.2.4 The Parameterized ROM Megafunction (lpm_rom)

The `lpm_rom` block is parameterized ROM with separate input and output clocks. We have used this megafunction for the designs `fun_text`, p. 30 and `darom`, p. 196.

The `lpm_rom` block can also be used with older device families, e.g., Flex 10K. Altera translates for newer devices like Cyclone II the `lpm_rom` in the `altsyncram` megafunction block. But the `altsyncram` is not supported for Flex 10K that is used on the popular UP2 boards.

Altera recommends instantiating this function as described in "Using the MegaWizard Plug-In Manager" in the Quartus II help.

The port names and order for Verilog HDL prototype are:

```
module lpm_rom ( address, inclock, outclock, memenab,
                 q);
```

The VHDL component declaration is shown below:

```
COMPONENT lpm_rom
  GENERIC (LPM_WIDTH    : POSITIVE;
           LPM_TYPE     : STRING := "L_ROM";
           LPM_WIDTHAD  : POSITIVE;
           LPM_NUMWORDS : POSITIVE;
           LPM_FILE     : STRING;
           LPM_ADDRESS_CONTROL : STRING := "REGISTERED";
           LPM_OUTDATA  : STRING := "REGISTERED";
           LPM_HINT     : STRING := "UNUSED");
 PORT(address : IN STD_LOGIC_VECTOR(LPM_WIDTHAD-1 DOWNTO 0);
      inclock : IN STD_LOGIC := '1';
      outclock : IN STD_LOGIC := '1';
      memenab  : IN STD_LOGIC := '1';
      q        : OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0)
      );
END COMPONENT;
```

Ports

The following table displays all input ports of `lpm_rom`:

Port name	Required	Description	Comments
`address`	Yes	Address input to the memory	Input port `LPM_WIDTHAD` wide
`inclock`	No	Clock for input registers	The `address` port is synchronous (registered) when the `inclock` port is connected, and is asynchronous (registered) when the `inclock` port is not connected.
`outclock`	No	Clock for output registers	The addressed memory content-to-`q` response is synchronous when the `outclock` port is connected, and is asynchronous when it is not connected.
`memenab`	No	Memory enable input	High = data output on `q`, Low = high-impedance outputs

The following table displays all output ports of `lpm_rom`:

Port Name	Required	Description	Comments
`q`	Yes	Output of memory	Output port `LPM_WIDTH` wide

Parameters

The following table shows the parameters of the `lpm_rom` component:

Parameter	Type	Required	Description
`LPM_WIDTH`	Integer	Yes	Width of the `q` port.
`LPM_WIDTHAD`	Integer	Yes	Width of the `address` port. `LPM_WIDTHAD` should be (but is not required to be) equal to $\log_2$(`LPM_NUMWORDS`). If `LPM_WIDTHAD` is too small, some memory locations will not be addressable. If it is too large, addresses that are too high will return undefined logic levels.
`LPM_NUMWORDS`	Integer	Yes	Number of words stored in memory. In general, this value should be (but is not required to be) $2^{\texttt{LPM\_WIDTHAD}} - 1 < \texttt{LPM\_NUMWORDS} \leq 2^{\texttt{LPM\_WIDTHAD}}$. If omitted, the default is $2^{\texttt{LPM\_WIDTHAD}}$.
`LPM_FILE`	String	No	Name of the `Memory Initialization File` (*.mif) or `Hexadecimal` (Intel-format) File (*.hex) containing ROM initialization data (<filename>), or `UNUSED`.
`LPM_ADDRESS_CONTROL`	String	No	Values are `REGISTERED`, `UNREGISTERED`, and `UNUSED`. Indicates whether the address port is registered. If omitted, the default is `REGISTERED`.
`LPM_OUTDATA`	String	No	Values are `REGISTERED`, `UNREGISTERED`, and `UNUSED`. Indicates whether the q and eq ports are registered. If omitted, the default is `REGISTERED`.
`LPM_HINT`	String	No	Allows you to specify Altera-specific parameters in VHDL design files. The default is `UNUSED`.
`LPM_TYPE`	String	No	Identifies the LPM entity name in VHDL design files.

Note that for Verilog LPM 220 synthesizable code (i.e., `220model.v`) the following parameter ordering applies: `lpm_type`, `lpm_width`, `lpm_widthad`, `lpm_numwords`, `lpm_address_control`, `lpm_outdata`, `lpm_file`, `lpm_hint`.

Function

The following table shows the *synchronous* read from memory behavior of `lpm_rom`:

OUTCLOCK	MEMENAB	Function
X	L	`q` output is high impedance (memory not enabled)
⌊	H	No change in output
⌈	H	The output register is loaded with the contents of the memory location pointed to by `address`. `q` outputs the contents of the output register.

The output `q` is asynchronous and reflects the data in the memory to which `address` points. The following table shows the *asynchronous* memory operations behavior of `lpm_rom`:

MEMENAB	Function
L	`q` output is high-impedance (memory not enabled)
H	The memory location pointed to by `address` is read

Totally asynchronous memory operations occur when neither `inclock` nor `outclock` is connected. The output `q` is asynchronous and reflects the memory location pointed to by `address`. Since this totally asynchronous memory operation is only available with Flex 10K devices, but not with Cyclone II, we do not use this mode in our designs. Either input or output is registered in all of our designs that use memory blocks.

Resource Usage

The Megafunction `lpm_rom` uses one embedded cell per memory bit.

B.2.5 The Parameterized Divider Megafunction (lpm_divide)

Altera recommends that you use `lpm_divide` to replace all other types of divider functions, including old-style divide macrofunction. We have used this megafunction for the array divider designs p. 103.

Altera recommends instantiating this function as described in "Using the MegaWizard Plug-In Manager" in the Quartus II help.
The port names and order for Verilog HDL prototype are:

```
module lpm_divide ( quotient, remain, numer, denom,
                                  clock, clken, aclr )
```

The VHDL component declaration is shown below:

```
COMPONENT lpm_divide
   GENERIC ( LPM_WIDTHN: POSITIVE;
             LPM_WIDTHD: POSITIVE;
             LPM_NREPRESENTATION: STRING: = "UNSIGNED";
             LPM_DREPRESENTATION: STRING: = "UNSIGNED";
             LPM_TYPE: STRING :="LPM_DIVIDE";
```

```
            LPM_PIPELINE: INTEGER := 0;
            LPM_HINT: STRING:= "UNUSED";
          );
    PORT    ( numer: IN STD_LOGIC_VECTOR(LPM_WIDTHN-1 DOWNTO 0);
            denom: IN STD_LOGIC_VECTOR(LPM_WIDTHD-1 DOWNTO 0);
            clock, aclr: IN STD_LOGIC := '0';
            clken: IN STD_LOGIC := '1';
            quotient: OUT STD_LOGIC_VECTOR(LPM_WIDTHN-1 DOWNTO 0);
            remain: OUT STD_LOGIC_VECTOR(LPM_WIDTHD-1 DOWNTO 0)
          );
END COMPONENT;
```

Ports

The following table displays all input ports of `lpm_divide`:

Port name	Required	Description	Comments
`numer`	Yes	Numerator	Input port `LPM_WIDTHN` wide.
`denom`	Yes	Denominator	Input port `LPM_WIDTHD` wide.
`clock`	No	Clock input for pipelined usage.	You must connect the clock input if you set `LPM_PIPELINE` to a value other than 0.
`clken`	No	Clock enable for pipelined usage.	
`aclr`	No	Asynchronous clear signal.	The `aclr` port may be used at any time to reset the pipeline to all 0s asynchronously to the `clock` input.

The following table displays all output ports of `lpm_divide`:

Port Name	Required	Description	Comments
`quotient`	Yes	Output port `LPM_WIDTHN` wide.	You must use either the `quotient` or the `remain` ports.
`remain`	Yes	Output port `LPM_WIDTHD` wide.	You must use either the `quotient` or the `remain` ports.

Parameters

The following table shows the parameters of the `lpm_divide` component:

Parameter	Type	Required	Description
`LPM_WIDTHN`	Integer	Yes	Width of the `numer` and `quotient` port
`LPM_WIDTHD`	Integer	Yes	Width of the `denom` and `remain` port
`LPM_ _NREPRESENTATION`	String	No	Specifies whether the numerator is `SIGNED` or `UNSIGNED`. Only `UNSIGNED` is supported for now.
`LPM_ _DREPRESENTATION`	String	No	Specifies whether the denominator is `SIGNED` or `UNSIGNED`. Only `UNSIGNED` is supported for now.
`LPM_PIPELINE`	Integer	No	Specifies the number of clock cycles of latency associated with the `quotient` and `remain` outputs. A value of zero (0) indicates that no latency exists, and that a purely combinatorial function will be instantiated. If omitted, the default is 0 (nonpipelined). You cannot specify a value for the `LPM_PIPELINE` parameter that is higher than `LPM_WIDTHN`.
`LPM_TYPE`	String	No	Identifies the LPM entity name in VHDL design files.
`LPM_HINT`	String	No	Allows you to assign Altera-specific parameters in VHDL design files. The default is `UNUSED`.

You can pipeline a design by connecting the `clock` input and specifying the number of clock cycles of latency with the `LPM_PIPELINE` parameter.

Note that for Verilog LPM 220 synthesizable code (i.e., `220model.v`) the following parameter ordering applies: `lpm_type`, `lpm_widthn`, `lpm_widthd`, `lpm_nrepresentation`, `lpm_drepresentation`, `lpm_pipeline`.

B.2.6 The Parameterized RAM Megafunction (lpm_ram_dq)

The `lpm_ram_dq` block is parameterized RAM with separate input and output ports. The `lpm_ram_dq` block can also be used with older device families, e.g., Flex10K. Altera translates for newer devices like Cyclone II the `lpm_ram_dq` in the `altsyncram` megafunction block. But the `altsyncram` is not supported for Flex10K that is used on the popular UP2 boards.

We have used this megafunction for the design `trisc0`, p. 606.

Altera recommends instantiating this function as described in "Using the MegaWizard Plug-In Manager" in the Quartus II help.

The port names and order for Verilog HDL prototype are:

```
module lpm_ram_dq ( q, data, inclock, outclock, we, address);
```

The VHDL component declaration is shown below:

```
COMPONENT lpm_ram_dq
   GENERIC (LPM_WIDTH           : POSITIVE;
            LPM_WIDTHAD         : POSITIVE;
            LPM_NUMWORDS        : NATURAL   := 0;
            LPM_INDATA          : STRING    := "REGISTERED";
            LPM_ADDRESS_CONTROL : STRING    := "REGISTERED";
            LPM_OUTDATA         : STRING    := "REGISTERED";
            LPM_FILE            : STRING    := "UNUSED";
            LPM_TYPE            : STRING    := "LPM_RAM_DQ";
            LPM_HINT            : STRING    := "UNUSED" );
  PORT (data : IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
     address : IN STD_LOGIC_VECTOR(LPM_WIDTHAD-1 DOWNTO 0);
   inclock, outclock          : IN STD_LOGIC := '0';
         we  : IN STD_LOGIC;
          q  : OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0)
          );
END COMPONENT;
```

Ports

The following table displays all input ports of `lpm_ram_dq`:

Port name	Required	Description	Comments
`address`	Yes	Address input to the memory	Input port `LPM_WIDTHAD` wide
`data`	Yes	Data input to the memory	Input port `LPM_WIDTHAD` wide
`inclock`	No	Clock for input registers	The `address` port is synchronous (registered) when the `inclock` port is connected, and is asynchronous (registered) when the `inclock` port is not connected.
`outclock`	No	Clock for output registers	The addressed memory content-to-`q` response is synchronous when the `outclock` port is connected, and is asynchronous when it is not connected.
`we`	Yes	Memory enable input	Write enable input. Enables write operations to the memory when high.

The following table displays all output ports of `lpm_ram_dq`:

Port Name	Required	Description	Comments
`q`	Yes	Output of memory	Output port `LPM_WIDTH` wide

Parameters

The following table shows the parameters of the `lpm_ram_dq` component:

Parameter	Type	Required	Description
`LPM_WIDTH`	Integer	Yes	Width of the `q` port.
`LPM_WIDTHAD`	Integer	Yes	Width of the `address` port. `LPM_WIDTHAD` should be (but is not required to be) equal to $\log_2$(`LPM_NUMWORDS`). If `LPM_WIDTHAD` is too small, some memory locations will not be addressable. If it is too large, addresses that are too high will return undefined logic levels.
`LPM_NUMWORDS`	Integer	Yes	Number of words stored in memory. In general, this value should be (but is not required to be) $2^{\texttt{LPM\_WIDTHAD}} - 1 < \texttt{LPM\_NUMWORDS} \leq 2^{\texttt{LPM\_WIDTHAD}}$. If omitted, the default is $2^{\texttt{LPM\_WIDTHAD}}$.
`LPM_FILE`	String	No	Name of the `Memory Initialization File` (*.mif) or `Hexadecimal` (Intel-format) File (*.hex) containing ROM initialization data (<filename>), or `UNUSED`.
`LPM_ADDRESS_CONTROL`	String	No	Values are `REGISTERED`, `UNREGISTERED`, and `UNUSED`. Indicates whether the address port is registered. If omitted, the default is `REGISTERED`.
`LPM_OUTDATA`	String	No	Values are `REGISTERED`, `UNREGISTERED`, and `UNUSED`. Indicates whether the `q` and `eq` ports are registered. If omitted, the default is `REGISTERED`.
`LPM_HINT`	String	No	Allows you to specify Altera-specific parameters in VHDL design files. The default is `UNUSED`.
`LPM_TYPE`	String	No	Identifies the LPM entity name in the VHDL design files.

Note that for Verilog LPM 220 synthesizable code (i.e., `220model.v`) the following parameter ordering applies: `lpm_type`, `lpm_width`, `lpm_widthad`, `lpm_numwords`, `lpm_address_control`, `lpm_outdata`, `lpm_file`, `lpm_hint`.

Function

The following table shows the *synchronous* read and write memory behavior of `lpm_ram_dq`:

`inclock`	`we`	Function
X	-	No change (requires rising clock edge).
⎍	H	The memory location pointed to by `address` is loaded with `data`.
⎍	L	The memory location pointed to by `address` is read from the array. If `outclock` is not used, the read data appears at the outputs.

The following table shows the synchronous read from memory from memory operations behavior of `lpm_ram_dq`:

`outclock`	Function
–	No change
⎍	The memory location pointed to by `address` is read and written into the output register.

Totally asynchronous memory operations occur when neither `inclock` nor `outclock` is connected. The output `q` is asynchronous and reflects the memory location pointed to by `address`. Since this totally asynchronous memory operation is only available with Flex 10K devices, but not with Cyclone II, we do not use this mode in our designs. Either the input or output is registered in all of our designs that use memory blocks.

Resource Usage

The Megafunction `lpm_ram_dq` uses one embedded cell per memory bit.

C. 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
ALU	Arithmetic logic unit
AM	Amplitude modulation
AMBA	Advanced microprocessor bus architecture
AMD	Advanced Micro Devices, Inc.
ASCII	American standard code for information interchange
ASIC	Application-specific IC
AWGN	Additive white Gaussian noise
BCD	Binary coded decimal
BDD	Binary decision diagram
BLMS	Block LMS
BP	Bandpass
BRS	Base removal scaling
BS	Barrelshifter
CAE	Computer-aided engineering
CAM	Content addressable memory
CAST	Carlisle Adams and Stafford Tavares
CBC	Cipher block chaining
CBIC	Cell-based IC
CD	Compact disc
CFA	Common factor algorithm
CFB	Cipher feedback
CIC	Cascaded integrator comb
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
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
DES	Data encryption standard
DFT	Discrete Fourier transform
DHT	Discrete Hartley transform
DIF	Decimation in frequency
DIT	Decimation in time
DLMS	Delayed LMS
DMA	Direct memory access
DMIPS	Dhrystone MIPS
DMT	Discrete Morlet transform
DPLL	Digital PLL
DSP	Digital signal processing
DST	Discrete sine transform
DWT	Discrete wavelet transform
EAB	Embedded array block
ECB	Electronic code book
ECL	Emitter coupled logic
EDIF	Electronic design interchange format
EFF	Electronic Frontier Foundation
EPF	Altera FPGA family
EPROM	Electrically programmable ROM
ERA	Plessey FPGA family
ERNS	Eisenstein RNS
ESA	European Space Agency
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
FSF	Frequency sampling filter
FSK	Frequency shift keying
FSM	Finite state machine
GAL	Generic array logic
GF	Galois field
GNU	GNU's not Unix
GPP	General-purpose processor
GPR	General-purpose register
HB	Half-band filter
HI	High frequency
HDL	Hardware description language
HSP	Harris Semiconductor DSP ICs
IBM	International Business Machines (corporation)
IC	Integrated circuit
ICU	Instruction cache unit
IDCT	Inverse DCT
IDEA	International data encryption algorithm
IDFT	Inverse discrete Fourier transform
IEEE	Institute of Electrical and Electronics Engineers
IF	Inter frequency
IFFT	Inverse fast Fourier transform
IIR	Infinite impulse response
I-MOMS	Interpolating MOMS
INTT	Inverse NTT
IP	Intellectual property
I/Q	In-/Quadrature phase
ISA	Instruction set architecture
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
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	Lowpass
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
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
μP	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
NP	Nonpolynomial complex problem
NRE	Nonreccurring engineering costs
NTT	Number theoretic transform
OFB	Open feedback (mode)
O-MOMS	Optimal MOMS
PAM	Pulse-amplitude modulated
PC	Personal computer
PCI	Peripheral component interconnect
PD	Phase detector
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
PREP	Programmable Electronic Performance (cooperation)
PRNS	Polynomial RNS
PROM	Programmable ROM
PSK	Phase shift keying
PSM	Programmable state machine
QDFT	Quantized DFT
QLI	Quick look-in
QFFT	Quantized FFT
QMF	Quadrature mirror filter
QRNS	Quadratic RNS
QSM	Quarter square multiplier
RAM	Random-access memory
RC	Resistor/capacity
RF	Radio frequency
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
SG	Stochastic gradient
SIMD	Single instruction multiple data
SLMS	Signed LMS
SM	Signed magnitude
SNR	Signal-to-noise ratio
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
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
TTL	Transistor transistor logic
TVP	True vector processor
UART	Universal asynchronous receiver/transmitter
VCO	Voltage-control oscillator
VHDL	VHSIC hardware description language

VHSIC	Very-high-speed integrated circuit
VLIW	Very long instruction word
VLSI	Very large integrated ICs
WDT	Watchdog timer
WFTA	Winograd Fourier transform algorithm
WSS	Wide sense stationary
XC	Xilinx FPGA family
XNOR	Exclusive NOR gate
YACC	Yet another compiler-compiler

D. CD-ROM File: "1readme.ps"

The accompanying CD-ROM includes

- A full version of the Quartus II software
- Altera datasheets for Cyclone II devices
- All VHDL/Verilog design examples and utility programs and files

To install the Quartus II 6.0 web edition software first read the licence agreement carefully. Since the Quartus II 6.0 web edition software uses many other tools (e.g., GNU, Berkeley Tools, SUN microsystems tool, etc.) you need to agree to their licence agreements too before installing the software. To install the software start the self-extracting file `quartusii_60_web_edition.exe` on the CD-ROM in the `Altera` folder. After the installation the user must register the software through Altera's web page at `www.altera.com` in order to get a permanent licence key. Otherwise the temporary licence key expires after the 30-day grace period and the software will no longer run. Altera frequently update the Quartus II software to support new devices 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. Altera's University program now delivers the files via download, which can take long time with a 56 Kbit/s MODEM.

The design examples for the book are located in the directories `book3e/vhdl` and `book3e/verilog` for the VHDL and Verilog examples, respectively. These directories contain, for each example, the following four files:

- The VHDL or Verilog source code (`*.vhd` and `*.v`)
- The Quartus project files (`*.qpf`)
- The Quartus setting files (`*.qsf`)
- The Simulator wave form file (`*.vwf`)

For the design `fun_graf`, the block design file (`*.bdf`) is included in `book3e/vhdl`. For the examples that utilize M4Ks (i.e., `fun_text`, `darom`, and `trisc0`), the memory initialization file (`*.mif`) can be found on the CD-ROM. To simplify the compilation and postprocessing, the source code directories include the additional (`*.bat`) files and Tcl scripts shown below:

File	Comment
`qvhdl.tcl`	Tcl script to compile all design examples. Note that the device can be changed from Cyclone II to Flex, Apex or Stratix just by changing the comment sign # in column 1 of the script.
`qclean.bat`	Cleans all temporary Quartus II compiler files, but not the report files (`*.map.rpt`), the timing analyzer output files (`*.tan.rpt`), and the project files `*.qpf` and `*.qsf`.
`qveryclean.bat`	Cleans all temporary compiler files, *including* all report files (`*.rep`) and project files.

Use the `DOS` prompt and type

```
quartus_sh -t qvhdl.tcl > qvhdl.txt
```

to compile all design examples and then `qclean.bat` to remove the unnecessary files. The Tcl script `qvhdl.tcl` is included on the CD. The Tcl script language developed by the Berkeley Professor John Ousterhout [346, 347, 348] (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 looks almost the same.

Two search procedures (`show_fmax` and `show_resources`) are used within the Tcl script `qvhdl.tcl` to display resources and `Registered Performance`. The script includes all settings and also alternative device definitions. The protocol file `qvhdl.txt` has all the useful synthesis data. For the trisc0 processor, for instance, the list for the Cyclone II device `EP2C35F672C6` is:

```
....
--------------------------------------------------
trisc0 fmax: 115.65 MHz ( period = 8.647 ns )
trisc0 LEs: 198 / 33,216 ( < 1 % )
trisc0 M4K bits: 5,120 / 483,840 ( 1 % )
trisc0 DSP blocks: 1 / 70 ( 1 % )
--------------------------------------------------
....
```

The results for all examples are summarized in Table B.1, p. 731.

Other devices are prespecified and include the `EPF10K20RC240-4` and `EPF10K70RC240-4` from the UP1 and UP2 University boards, the `EP20K200EFC484-2X` from the Nios development boards, and three devices from other DSP boards available from Altera, i.e., the `EP1S10F484C5`, `EP1S25F780C5`, and `EP2S60F1020C4ES`.

Using Compilers Other Then Quartus II

Synopsys FPGA_CompilerII

The main advantage of using the FPGA_CompilerII (FC2) from Synopsys was that it was possible to synthesize examples for other devices like Xilinx, Vantis, Actel, or QuickLogic with the same tool. The Tcl scripts `vhdl.fc2`, and `verilog.fc2`,

respectively, were provided the necessary commands for the shell mode of `FC2`, i.e., `fc2_shell` in the second edition of the book [57]. Synopsys, however, since 2006 no longer supports the FPGA_CompilerII and it is therefore not a good idea to use the compiler anymore since the newer devices can not be selected.

Model Technology

By using the synthesizable public-domain models provided by the EDIF organization (at `www.edif.org`), it is also possible to use other VHDL/Verilog simulators then Quartus II.

Using MTI and VHDL. For VHDL, the two files `220pack.vhd` and `220model.vhd` must first be compiled. For the ModelSim simulator `vsim` from Model Technology Inc., the script `mti_vhdl.do` can be used for device-independent compilation and simulation of the design examples. The script is shown below:

```
#-----------------------------------------------------------
# Model Technology VHDL compiler script for the book
# Digital Signal Processing with FPGAs (3.edition)
# Author-EMAIL: Uwe.Meyer-Baese@ieee.org
#-----------------------------------------------------------

echo Create Library directory lpm
vlib lpm

echo Compile lpm package.
vcom -work lpm -explicit -quiet 220pack.vhd 220model.vhd

echo Compile chapter 1 entitys.
vcom -work lpm -quiet example.vhd fun_text.vhd

echo Compile chapter 2 entitys.
vcom -work lpm -explicit -quiet add1p.vhd add2p.vhd
vcom -work lpm -explicit -quiet add3p.vhd mul_ser.vhd
vcom -work lpm -explicit -quiet cordic.vhd

echo Compile chapter 3 components.
vcom -work lpm -explicit -quiet case3.vhd case5p.vhd
vcom -work lpm -explicit -quiet case3s.vhd
echo Compile chapter 3 entitys.
vcom -work lpm -explicit -quiet fir_gen.vhd fir_srg.vhd
vcom -work lpm -explicit -quiet dafsm.vhd darom.vhd
vcom -work lpm -explicit -quiet dasign.vhd dapara.vhd

echo Compile chapter 4 entitys.
vcom -work lpm -explicit -quiet iir.vhd iir_pipe.vhd
vcom -work lpm -explicit -quiet iir_par.vhd

echo Compile chapter 5 entitys.
vcom -work lpm -explicit -quiet cic3r32.vhd cic3s32.vhd
vcom -work lpm -explicit -quiet db4poly.vhd db4latti.vhd

echo Compile chapter 6 entitys.
```

```
vcom -work lpm -explicit -quiet rader7.vhd ccmul.vhd
vcom -work lpm -explicit -quiet bfproc.vhd

echo Compile chapter 7 entitys.
vcom -work lpm -explicit -quiet rader7.vhd ccmul.vhd
vcom -work lpm -explicit -quiet bfproc.vhd

echo Compile 2. edition entitys.
vcom -work lpm -explicit -quiet div_res.vhd div_aegp.vhd
vcom -work lpm -explicit -quiet fir_lms.vhd fir6dlms.vhd

echo Compile 3. edition entitys from chapter 2.
vcom -work lpm -explicit -quiet cmul7p8.vhd arctan.vhd
vcom -work lpm -explicit -quiet ln.vhd sqrt.vhd

echo Compile 3. edition entitys from chapter 5.
vcom -work lpm -explicit -quiet rc_sinc.vhd farrow.vhd
vcom -work lpm -explicit -quiet cmoms.vhd

echo Compile 3. edition entitys from chapter 9.
vcom -work lpm -explicit -quiet reg_file.vhd trisc0.vhd
```

Start the ModelSim simulator and then type

```
do mti_vhdl.do
```

to execute the script.

Using MTI and Verilog. Using the Verilog interface with the `lpm` library from EDIF, i.e., `220model.v`, needs some additional effort. When using `220model.v` it is necessary to specify *all* ports in the Verilog `lpm` components. There is an extra directory `book3e/verilog/mti`, which provides the design examples with a full set of `lpm` port specifications. The designs use

```
‘\include "220model.v"
```

at the beginning of each Verilog file to include the `lpm` components, if necessary. Use the script `mti_v1.csh` and `mti_v2.csh` to compile all Verilog design examples with Model Technology's `vcom` compiler.

In order to load the memory initialization file (`*.mif`), it is required to be familiar with the programming language interface (PLI) of the Verilog 1364-1995 IEEE standard (see LRM Sect. 17, p. 228 ff). With this powerful PLI interface, conventional `C` programs can be dynamically loaded into the Verilog compiler. In order to generate a dynamically loaded object from the program `convert_hex2ver.c`, the path for the include files `veriuser.h` and `acc_user.h` must be specified. Use `-I` when using the `gcc` or `cc` compiler under SUN Solaris. Using, for instance, the `gcc` compiler under SUN Solaris for the Model Technology Compiler, the following commands are used to produce the shared object:

```
gcc -c -I/<install_dir>/modeltech/include convert_hex2ver.c
ld -G -B symbolic -o convert_hex2ver.sl convert_hex2ver.o
```

By doing so, `ld` will generate a warning "Symbol referencing errors," because all symbols are first resolved within the shared library at link time, but these warnings can be ignored.

It is then possible to use these shared objects, for instance, with Model Technology's `vsim` in the first design example `fun_text.v`, with

```
vsim  -pli convert_hex2verl.sl  lpm.fun_text
```

To learn more about PLIs, check out the Verilog IEEE standard 1364-1995, or the vendor's user manual of your Verilog compiler.

We can use the script `mti_v1.do` to compile all Verilog examples with MTI's `vlog`. Just type

```
do mti_v1.do
```

in the ModelTech command line. But `vlog` does not perform a check of the correct component port instantiations or shared objects. A second script, `mti_v2.do`, can be used for this purpose. Start the `vsim` simulator (without loading a design) and execute the `DO` file with

```
do mti_v2.do
```

to perform the check for all designs.

Using Xilinx ISE The conversion of designs from Altera Quartus II to Xilinx ISE seems to be easy if we use standard HDL. Unfortunately there a couple of issues that needs to be addressed. We assume that the ModelTech simulation environment and the web version (i.e., no core generation) is used. We like to discuss in the following a couple of items that address the ISE/ModelTech design entry. We describe the Xilinx ISE 6.2 web edition and ModelTech 5.7g version.

1) The Xilinx simulation with timing ("Post-Place & Route") uses a bitwise simulation model on the LUT level. Back annotations are only done for the I/O ports, and are ALL from type `standard_logic` or `standard_logic_vector`. In order to match the behavior and the simulation with timing we therefore need to use only the `standard_logic` or `standard_logic_vector` data type for I/O. As a consequence no integers, generic, or custom I/O data type, (e.g., like the subtype byte see `cordic.vhd`) can be used.

2) The ISE software supports the development of testbenches with the "Test Bench Waveform." Use `New Source...` under the `Project` menu. This waveform will give you a quick way to generate a testbench that is used by ModelTech, both for behavior as well as simulation with timing. There are some benefits and drawbacks with the testbencher. For instance, you can not assign negative integers in the waveforms, you need to build the two's complement, i.e., equivalent unsigned number by hand.

3) If you have feedback, you need to initialize the register to zero in your HDL code. You can not do this with the testbencher: for instance, ModelTech initialize all integer signals to the smallest value, i.e., -128 for a 8-bit number, if you add two integers, the result will be $-128 - 128 = -256 < -128$ and ModelTech will stop and report an overflow. Some designs, e.g., `cordic`, `cic3r32`, `cic3s32`, only work correctly in behavior simulation if all integers are changed to `standard_logic_vector` data type. Changing I/O ports alone and using the conversion function does not always guarantee correct simulation results.

4) Simulation with timing usually needs one clock cycle more than behavior code until all logic is settled. The input stimuli should therefore be zero in the first clock cycle (ca. 100 ns) and, if you want to match behavior and timing simulation, and the design uses a (small) FSM for control, you need to add a synchronous or asynchronous reset. You need to do this for the following 2/e designs: `dafsm, dadrom, dasign, db4latti, db4poly, div_aegp, div_res, iir_par, mul_ser, rader7.`
Just add a control part for the FSM like this:

```
-- IF rising_edge(clk) THEN        -- Synchronous reset
--     IF reset = '1' THEN
--        state <= s0;
--     ELSE
    IF reset = '1' THEN            -- Asynchronous reset
      state <= s0;
    ELSIF rising_edge(clk) THEN
      CASE state IS
      WHEN s0 =>                   -- Initialization step
...
```

Although at first glance this synchronous or asynchronous control seems to be cheap because the FSM is small, we need to keep in mind that, if the `reset` is active, all signals that are assigned in the state s0 of the FSM need to be preserved with their initial state value. The following table shows the synthesis results for the three different reset styles for the design file `dafsm.vhd` (small distribute arithmetic state machine):

Reset style	Performance/ns	4-input LUT	Gates
No reset (original code)	3.542	20	339
synchronous	3.287	29	393
asynchronous	3.554	29	393

Designs with reset usually have a higher LUT and gate count. Depending on the design, synchronous or asynchronous reset can also have a (small) influence on performance.

5) Back annotation is only done for I/O ports. If we want to monitor internal nets, we can try to find the appropriate net name in the `*_timesim.vhd` file, but that is quite complicated and may change in the next compiler run. A better idea is to introduce additional test outputs, see, for instance, `fir_lms.vhd` for `f1_out` and `f2_out`. In the behavioral (but not in the timing) simulation internal test signals and variables can be monitored. Modify the `*.udo` file and add, for instance for the `fir_srg_tb.vhd` file, `add wave /fir_srg_tb/uut/tap` to the testbench.

3) There are a couple of nice features in the Xilinx ISE package too: there is no need for special `lpm` blocks to use the internal resources for multiplier, shifter, RAMs or ROMs. Some other features are:

a) ISE converts a shift register in a single CLB-based shift register. This can save some resources.

b) Multipliers can be implemented with LUTs only, including block multipliers (if available) or even pipelined LUTs, which is done via pipeline retiming. Just right click on the `Synthesize-XST` menu in the `Processes`, select `HDL Options` under `Process Properties` and the last entry is the multiplier style. But note that for pipelined LUT design the additional register must be placed at the *output* of the multiplier. Pipeline retiming is not done if the additional registers are at the inputs. You need about $\log_2(B)$ additional registers to have good timing (see Chap. 2 on pipeline multiplier). This has an impact on the `Registered Performance`, LUT usage, and gates as the following table shows for the `fir_gen.vhd` example, i.e., length 4 programmable FIR filter (from Chap. 3):

Synthesis style	Speed in ns	4-input LUT	mul. blocks	Gates
Block multiplier	9.838	57	4	17552
LUT (no pipeline)	15.341	433	0	6114
LUT (3 stage pipeline)	6.762	448	0	9748

For this multiplier size (9 bit) the pipelined LUT seems to be attractive, both for speed as well as gate count. If the number of LUTs is limited, the block multiplier provides the next best alternative.

c) If you follow the recommended style the Xilinx software synthesis tool (see XST manual and ISE help "Inferring BlockRAM in VHDL") maps your HDL code to the block RAM (see, `fun_text.vhd`). If the table is small, the ISE auto option selects the LUT-based implementation for a ROM table (see `darom.vhd`). You can also initialize the table in the HDL code and use it as a ROM. Please see the XST manual Chap. 3, "FPGA Optimization" for details on ROM implementation. There are some limitations that apply to the initialization of BlockRAMs (see, XST Chap. 2)

Utility Programs and Files

A couple of extra utility programs are also included on the CD-ROM[1] and can be found in the directory `book3e/util`:

File	Description
`sine3e.exe`	Program to generate the MIF files for the function generator in Chap. 1
`csd3e.exe`	Program to find the canonical signed digit representation of integers or fractions as used in Chap. 2
`fpinv3e.exe`	Program to compute the floating-point tables for reciprocals as used in Chap. 2
`dagen.exe`	Program to generate the VHDL code for the distributed arithmetic files used in Chap. 3
`ragopt.exe`	Program to compute the reduced adder graph for constant-coefficient filters as used in Chap. 3. It has 10 predefined lowpass and half-band filters. The program uses a MAG cost table stored in the file `mag14.dat`
`cic.exe`	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 applications and should therefore run on Windows 95 or higher. The DOS script `Testall.bat` produces the examples used in the book.

Also under `book3e/util` we find the following utility files:

[1] You need to copy the programs to your harddrive first; you can not start them from the CD directly since the program write out the results in text files.

File	Description
quickver.pdf	Quick reference card for Verilog HDL from QUALIS
quickvhd.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
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 book3e/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.

Microprocessor Project Files and Programs

All microprocessor-related tools and documents can be found in the book3e/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 is an infix calculator 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 book3e/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.

Index