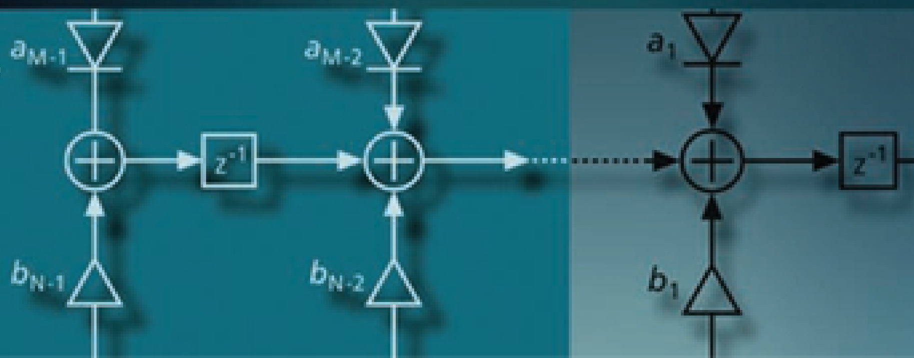
ROGER WOODS

JOHN McALLISTER

GAYE LIGHTBODY

YING YI

FPGA-based Implementation of Signal Processing Systems



FPGA-based Implementation of Signal Processing Systems

Roger Woods

Queen's University, Belfast, UK

John McAllister

Queen's University, Belfast, UK

Gaye Lightbody

University of Ulster, UK

Ying Yi

University of Edinburgh, UK

A John Wiley and Sons, Ltd., Publication

FPGA-based Implementation of Signal Processing Systems

FPGA-based Implementation of Signal Processing Systems

Roger Woods

Queen's University, Belfast, UK

John McAllister

Queen's University, Belfast, UK

Gaye Lightbody

University of Ulster, UK

Ying Yi

University of Edinburgh, UK

A John Wiley and Sons, Ltd., Publication

This edition first published 2008
© 2008 John Wiley & Sons, Ltd

Registered office
John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

For details of our global editorial offices, for customer services and for information about how to apply for permission to reuse the copyright material in this book please see our website at www.wiley.com.

The right of the author to be identified as the author of this work has been asserted in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except as permitted by the UK Copyright, Designs and Patents Act 1988, without the prior permission of the publisher.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The publisher is not associated with any product or vendor mentioned in this book. This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold on the understanding that the publisher is not engaged in rendering professional services. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

Library of Congress Cataloging-in-Publication Data

Wood, Roger.
FPGA-based implementation of complex signal processing systems / Roger Wood ... [et al.].
p. cm.
Includes bibliographical references and index.
ISBN 978-0-470-03009-7 (cloth)
1. Signal processing–Digital techniques. I. Title.
TK5102.5.W68 2008
621.382′2–dc22

2008035242

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library.

ISBN: 978-0-470-03009-7

Typeset by Laserwords Private Limited, Chennai, India
Printed and bound in Great Britain by Antony Rowe Ltd, Chippenham, Wiltshire

To Rose and Paddy

Contents

About the Authors **xv**

Preface **xvii**

1 Introduction to Field-programmable Gate Arrays **1**

1.1 Introduction 1
1.1.1 Field-programmable Gate Arrays 1
1.1.2 Programmability and DSP 3
1.2 A Short History of the Microchip 4
1.2.1 Technology Offerings 6
1.3 Influence of Programmability 7
1.4 Challenges of FPGAs 9
References 10

2 DSP Fundamentals **11**

2.1 Introduction 11
2.2 DSP System Basics 12
2.3 DSP System Definitions 12
2.3.1 Sampling Rate 14
2.3.2 Latency and Pipelining 15
2.4 DSP Transforms 16
2.4.1 Fast Fourier Transform 16
2.4.2 Discrete Cosine Transform (DCT) 18
2.4.3 Wavelet Transform 19
2.4.4 Discrete Wavelet Transform 19
2.5 Filter Structures 20
2.5.1 Finite Impulse Response Filter 20
2.5.2 Correlation 23
2.5.3 Infinite Impulse Response Filter 23
2.5.4 Wave Digital Filters 24
2.6 Adaptive Filtering 27
2.7 Basics of Adaptive Filtering 27
2.7.1 Applications of Adaptive Filters 28
2.7.2 Adaptive Algorithms 30

2.7.3 LMS Algorithm 31
2.7.4 RLS Algorithm 32
2.8 Conclusions 35
References 35

3 Arithmetic Basics **37**

3.1 Introduction 37
3.2 Number Systems 38
3.2.1 Number Representations 38
3.3 Fixed-point and Floating-point 41
3.3.1 Floating-point Representations 41
3.4 Arithmetic Operations 43
3.4.1 Adders and Subtracters 43
3.4.2 Multipliers 45
3.4.3 Division 47
3.4.4 Square Root 48
3.5 Fixed-point versus Floating-point 52
3.6 Conclusions 55
References 55

4 Technology Review **57**

4.1 Introduction 57
4.2 Architecture and Programmability 58
4.3 DSP Functionality Characteristics 59
4.4 Processor Classification 61
4.5 Microprocessors 62
4.5.1 The ARM Microprocessor Architecture Family 63
4.6 DSP Microprocessors (DSPμs) 64
4.6.1 DSP Micro-operation 66
4.7 Parallel Machines 67
4.7.1 Systolic Arrays 67
4.7.2 SIMD Architectures 69
4.7.3 MIMD Architectures 73
4.8 Dedicated ASIC and FPGA Solutions 74
4.9 Conclusions 75
References 76

5 Current FPGA Technologies **77**

5.1 Introduction 77
5.2 Toward FPGAs 78
5.2.1 Early FPGA Architectures 80
5.3 Altera FPGA Technologies 81
5.3.1 MAX® 7000 FPGA Technology 83
5.3.2 Stratix® III FPGA Family 85
5.3.3 Hardcopy® Structured ASIC Family 92
5.4 Xilinx FPGA Technologies 93
5.4.1 Xilinx Virtex™-5 FPGA Technologies 94

5.5 Lattice® FPGA Families 103
5.5.1 Lattice® ispXPLD 5000MX Family 103
5.6 Actel® FPGA Technologies 105
5.6.1 Actel® ProASIC^{PLUS} FPGA Technology 105
5.6.2 Actel® Antifuse SX FPGA Technology 106
5.7 Atmel® FPGA Technologies 108
5.7.1 Atmel® AT40K FPGA Technologies 108
5.7.2 Reconfiguration of the Atmel® AT40K FPGA Technologies 109
5.8 General Thoughts on FPGA Technologies 110
References 110

6 Detailed FPGA Implementation Issues 111

6.1 Introduction 111
6.2 Various Forms of the LUT 112
6.3 Memory Availability 115
6.4 Fixed Coefficient Design Techniques 116
6.5 Distributed Arithmetic 117
6.6 Reduced Coefficient Multiplier 120
6.6.1 RCM Design Procedure 122
6.6.2 FPGA Multiplier Summary 125
6.7 Final Statements 125
References 125

7 Rapid DSP System Design Tools and Processes for FPGA 127

7.1 Introduction 127
7.2 The Evolution of FPGA System Design 128
7.2.1 Age 1: Custom Glue Logic 128
7.2.2 Age 2: Mid-density Logic 128
7.2.3 Age 3: Heterogeneous System-on-chip 129
7.3 Design Methodology Requirements for FPGA DSP 129
7.4 System Specification 129
7.4.1 Petri Nets 129
7.4.2 Process Networks (PN) and Dataflow 131
7.4.3 Embedded Multiprocessor Software Synthesis 132
7.4.4 GEDAE 132
7.5 IP Core Generation Tools for FPGA 133
7.5.1 Graphical IP Core Development Approaches 133
7.5.2 Synplify DSP 134
7.5.3 C-based Rapid IP Core Design 134
7.5.4 MATLAB®-based Rapid IP Core Design 136
7.5.5 Other Rapid IP Core Design 136
7.6 System-level Design Tools for FPGA 137
7.6.1 Compaan 137
7.6.2 ESPAM 137
7.6.3 Daedalus 138
7.6.4 Koski 140
7.7 Conclusion 140
References 141

8 Architecture Derivation for FPGA-based DSP Systems 143

8.1 Introduction 143
8.2 DSP Algorithm Characteristics 144
8.2.1 Further Characterization 145
8.3 DSP Algorithm Representations 148
8.3.1 SFG Descriptions 148
8.3.2 DFG Descriptions 149
8.4 Basics of Mapping DSP Systems onto FPGAs 149
8.4.1 Retiming 150
8.4.2 Cut-set Theorem 154
8.4.3 Application of Delay Scaling 155
8.4.4 Calculation of Pipelining Period 158
8.5 Parallel Operation 161
8.6 Hardware Sharing 163
8.6.1 Unfolding 163
8.6.2 Folding 165
8.7 Application to FPGA 169
8.8 Conclusions 169
References 169

9 The IRIS Behavioural Synthesis Tool 171

9.1 Introduction of Behavioural Synthesis Tools 172
9.2 IRIS Behavioural Synthesis Tool 173
9.2.1 Modular Design Procedure 174
9.3 IRIS Retiming 176
9.3.1 Realization of Retiming Routine in IRIS 177
9.4 Hierarchical Design Methodology 179
9.4.1 White Box Hierarchical Design Methodology 180
9.4.2 Automatic Implementation of Extracting Processor Models from Previously Synthesized Architecture 181
9.4.3 Hierarchical Circuit Implementation in IRIS 184
9.4.4 Calculation of Pipelining Period in Hierarchical Circuits 185
9.4.5 Retiming Technique in Hierarchical Circuits 188
9.5 Hardware Sharing Implementation (Scheduling Algorithm) for IRIS 190
9.6 Case Study: Adaptive Delayed Least-mean-squares Realization 199
9.6.1 High-speed Implementation 200
9.6.2 Hardware-shared Designs for Specific Performance 205
9.7 Conclusions 207
References 207

10 Complex DSP Core Design for FPGA 211

10.1 Motivation for Design for Reuse 212
10.2 Intellectual Property (IP) Cores 213
10.3 Evolution of IP Cores 215
10.3.1 Arithmetic Libraries 216
10.3.2 Fundamental DSP Functions 218
10.3.3 Complex DSP Functions 219
10.3.4 Future of IP Cores 219

10.4 Parameterizable (Soft) IP Cores 220
10.4.1 Identifying Design Components Suitable for Development as IP 222
10.4.2 Identifying Parameters for IP Cores 223
10.4.3 Development of Parameterizable Features Targeted to FPGA Technology 226
10.4.4 Application to a Simple FIR Filter 228
10.5 IP Core Integration 231
10.5.1 Design Issues 231
10.5.2 Interface Standardization and Quality Control Metrics 232
10.6 ADPCM IP Core Example 234
10.7 Current FPGA-based IP Cores 238
10.8 Summary 240
References 241

11 Model-based Design for Heterogeneous FPGA 243

11.1 Introduction 243
11.2 Dataflow Modelling and Rapid Implementation for FPGA DSP Systems 244
11.2.1 Synchronous Dataflow 245
11.2.2 Cyclo-static Dataflow 246
11.2.3 Multidimensional Synchronous Dataflow 246
11.2.4 Dataflow Heterogeneous System Prototyping 247
11.2.5 Partitioned Algorithm Implementation 247
11.3 Rapid Synthesis and Optimization of Embedded Software from DFGs 249
11.3.1 Graph-level Optimization 250
11.3.2 Graph Balancing Operation and Optimization 250
11.3.3 Clustering Operation and Optimization 251
11.3.4 Scheduling Operation and Optimization 253
11.3.5 Code Generation Operation and Optimization 253
11.3.6 DFG Actor Configurability for System-level Design Space Exploration 254
11.3.7 Rapid Synthesis and Optimization of Dedicated Hardware from DFGs 254
11.3.8 Restricted Behavioural Synthesis of Pipelined Dedicated Hardware Architectures 255
11.4 System-level Modelling for Heterogeneous Embedded DSP Systems 257
11.4.1 Interleaved and Block Actor Processing in MADF 258
11.5 Pipelined Core Design of MADF Algorithms 260
11.5.1 Architectural Synthesis of MADF Configurable Pipelined Dedicated Hardware 261
11.5.2 WBC Configuration 262
11.6 System-level Design and Exploration of Dedicated Hardware Networks 263
11.6.1 Design Example: Normalized Lattice Filter 264
11.6.2 Design Example: Fixed Beamformer System 266
11.7 Summary 269
References 269

12 Adaptive Beamformer Example 271

12.1 Introduction to Adaptive Beamforming 271
12.2 Generic Design Process 272
12.3 Adaptive Beamforming Specification 274
12.4 Algorithm Development 276
12.4.1 Adaptive Algorithm 277
12.4.2 RLS Implementation 278

12.4.3 RLS Solved by QR Decomposition 278
12.4.4 Givens Rotations Used for QR Factorization 280
12.5 Algorithm to Architecture 282
12.5.1 Dependence Graph 283
12.5.2 Signal Flow Graph 283
12.5.3 Systolic Implementation of Givens Rotations 285
12.5.4 Squared Givens Rotations 287
12.6 Efficient Architecture Design 287
12.6.1 Scheduling the QR Operations 290
12.7 Generic QR Architecture 292
12.7.1 Processor Array 293
12.8 Retiming the Generic Architecture 301
12.8.1 Retiming QR Architectures 305
12.9 Parameterizable QR Architecture 307
12.9.1 Choice of Architecture 307
12.9.2 Parameterizable Control 309
12.9.3 Linear Architecture 310
12.9.4 Sparse Linear Architecture 310
12.9.5 Rectangular Architecture 316
12.9.6 Sparse Rectangular Architecture 316
12.9.7 Generic QR Cells 319
12.10 Generic Control 319
12.10.1 Generic Input Control for Linear and Sparse Linear Arrays 320
12.10.2 Generic Input Control for Rectangular and Sparse Rectangular Arrays 321
12.10.3 Effect of Latency on the Control Seeds 321
12.11 Beamformer Design Example 323
12.12 Summary 325
References 325

13 Low Power FPGA Implementation **329**

13.1 Introduction 329
13.2 Sources of Power Consumption 330
13.2.1 Dynamic Power Consumption 331
13.2.2 Static Power Consumption 332
13.3 Power Consumption Reduction Techniques 335
13.4 Voltage Scaling in FPGAs 335
13.5 Reduction in Switched Capacitance 337
13.6 Data Reordering 337
13.7 Fixed Coefficient Operation 338
13.8 Pipelining 339
13.9 Locality 343
13.10 Application to an FFT Implementation 344
13.11 Conclusions 348
References 348

14 Final Statements **351**

14.1 Introduction 351
14.2 Reconfigurable Systems 351
14.2.1 Relevance of FPGA Programmability 352
14.2.2 Existing Reconfigurable Computing 353
14.2.3 Realization of Reconfiguration 354
14.2.4 Reconfiguration Models 355
14.3 Memory Architectures 357
14.4 Support for Floating-point Arithmetic 358
14.5 Future Challenges for FPGAs 359
References 359

Index **361**

About the Authors

Roger Woods

Roger Woods has over 17 years experience in implementing complex DSP systems, both in ASIC and FPGA. He leads the Programmable Systems Laboratory at Queen's University (PSL@Q) which comprises 15 researchers and which is applying programmable hardware to DSP and telecommunicatins applications. The research specifically involves: developing design flows for heterogeneous platforms involving both multiprocessors and FPGAs; programmable solutions for programmable networks; design tools for FPGA IP cores; and low-power programmable DSP solutions. Roger has been responsible for developing a number of novel advanced chip demonstrators and FPGA solutions for image processing and digital filtering.

John McAllister

John McAllister is currently a Lecturer in the Programmable Systems Laboratory and System-on-Chip (SoC) Research Cluster at Queen's University Belfast investigating novel system, processor and IP core architectures, design methodologies and tools for programmable embedded DSP systems, with a particular focus on FPGA-centric processing architectures. He has numerous peer-reviewed publications in these areas.

Gaye Lightbody

Dr Gaye Lightbody received her MEng in Electrical and Electronic Engineering in 1995 and PhD in High-performance VLSI Architectures for Recursive Least-squares Adaptive Filtering in 2000, from the Queen's Univeristy of Belfast. During this time she worked as a research assistant before joining Amphion Semiconductor Limited (now Conexant Systems, Inc.) in January 2000 as a senior design engineer, developing ASIC and FPGA IP cores for the audio and video electronics industry. She returned to academia after five years in industry, taking up a position in the University of Ulster. Since then she has maintained an interest in VLSI design while broadening her activities into the area of Electrencephalography (EEG) evoked potential analysis and classification.

Ying Yi

Dr Ying Yi received the BSc degree in Computer and Application from Harbin Engineering University, Harbin, China, and the PhD degree from the Queen's University, Belfast, UK, in 1996 and 2003, respectively. She worked at the Wuhan Institute of Mathematical Engineering, China, as a

Software Engineer and then in research and development at the China Ship Research and Development Academy, Beijing, China. Currently, she is a Research Fellow at the University of Edinburgh, Edinburgh, UK. Her research interests include low-power reconfigurable SoC systems, compiler optimization techniques for reconfigurable architecture, architectural level synthesis optimization, and multiprocessor SoC.

Preface

DSP and FPGAs

Digital signal processing (DSP) is used in a very wide range of applications from high-definition TV, mobile telephony, digital audio, multimedia, digital cameras, radar, sonar detectors, biomedical imaging, global positioning, digital radio, speech recognition, to name but a few! The topic has been driven by the application requirements which have only been possible to realize because of development in silicon chip technology. Developing both programmable DSP chips and dedicated system-on-chip (SoC) solutions for these applications, has been an active area of research and development over the past three decades. Indeed, a class of dedicated microprocessors have evolved particularly targeted at DSP, namely DSP microprocessors or DSPμs.

The increasing costs of silicon technology have put considerable pressure on developing dedicated SoC solutions and means that the technology will be used increasingly for high-volume or specialist markets. An alternative is to use microprocessor style solutions such as microcontrollers, microprocessors and DSP micros, but in some cases, these offerings do not match well to the speed, area and power consumption requirements of many DSP applications. More recently, the field-programmable gate array (FPGA) has been proposed as a hardware technology for DSP systems as they offer the capability to develop the most suitable *circuit architecture* for the computational, memory and power requirements of the application in a similar way to SoC systems. This has removed the preconception that FPGAs are only used as 'glue logic' platform and more realistically shows that FPGAs are a collection of system components with which the user can create a DSP system. Whilst the prefabricated aspect of FPGAs avoids many of the deep submicron problems met when developing system-on-chip (SoC) implementations, the ability to create an efficient implementation from a DSP system description, remains a highly convoluted problem.

The book looks to address the implementation of DSP systems using FPGA technology by aiming the discussion at numerous levels in the FPGA implementation flow. First, the book covers *circuit* level, optimization techniques that allow the underlying FPGA fabric of localized memory in the form of lookup tables (LUTs) and flip-flops along with the logic LUT resource, to be used more intelligently. By considering the specific DSP algorithm operation in detail, it is shown that it is possible to map the system requirements to the underlying hardware, resulting in a more area-efficient, faster implementation. It is shown how the particular nature of some DSP systems such as DSP transforms (fast Fourier transform (FFT) and discrete cosine transform (DCT)) and fixed coefficient filtering, can be exploited to allow efficient LUT-based FPGA implementations.

Secondly, the issue of creating efficient circuit architecture from SFG representations is considered. It is clear that the development of a circuit architecture which efficiently uses the underlying resource to match the throughput requirements, will result in the most cost-effective solution. This requires the user to exploit the highly regular, highly computative, data-independent nature of DSP systems to produce highly parallel, pipelined circuit architectures for FPGA implementation. The availability of multiple, distributed logic resources and dedicated registers, make this type of

approach, highly attractive. Techniques are presented to allow the circuit architecture to be created with the necessary levels of parallelism and pipelining, resulting in the creation of highly efficient circuit architectures for the system under consideration.

Finally, as technology has evolved, FPGAs have now become a heterogeneous platform involving multiple hardware and software components and interconnection fabrics. It is clear that there is a strong desire for a true system-level design flow, requiring a much higher level system modelling language, in this case, *dataflow*. It is shown how the details of the language and approach must facilitate the kind of optimizations carried out to create the hardware functionality as outlined in the previous paragraph, but also to address system-level considerations such as interconnection and memory. This is a highly active area of research at present.

The book covers these three areas of FPGA implementation with a greater concentration on the latter two areas, namely that of the creation of the circuit architectures and the system level modelling, as these represent a more recent challenge; moreover, the circuit level optimization techniques have been covered in greater detail in many other places. It is felt that this represents a major differentiating factor between this book and other many other texts with a focus on FPGA implementation of DSP systems.

In all cases, the text looks to back up the description with the authors' experiences in implementing real DSP systems. A number of examples are covered in detail, including the development of an adaptive beamformer which gives a detailed description of the creation of an QR-based RLS filter. The design of an adaptive differential pulse-coded modulation (ADPCM) speech compression system is described. Throughout the text, finite impulse response (FIR) and infinite impulse response (IIR) filters are used to demonstrate the mapping and introduce retiming. The low-power optimizations are demonstrated using a FFT-based application and the development of hierarchical retiming, demonstrated using a wave digital filter (WDF).

In addition to the modelling and design aspect, the book also looks at the development of intellectual property (IP) cores as this has become a critical aspect in the creation of DSP systems. With the absence of relevant, high-level design tools, designers have resorted to creating reusable component blocks as a way of reducing the *design productivity gap*; this is the gap that has appeared between the technology and the designers' ability to use it efficiently. A chapter is given over to describing the creation of such IP cores and another chapter dedicated to the creation of a core for an important form of adaptive filtering, namely, recursive least-squares (RLS) filtering.

Audience

The book will be aimed at working engineers who are interested in using FPGA technology to its best in signal processing applications. The earlier chapters would be of interest to graduates and students completing their studies, taking the readers through a number of simple examples that show the various trade-offs when mapping DSP systems into FPGA hardware. The middle part of the book contains a number of illustrative, complex DSP systems that have been implemented using FPGAs and whose performance clearly illustrates the benefit of it use. These examples include: matrix multiplication, adaptive filtering systems for electronic support measures, wave digital filters, and adaptive beamformer systems based on RLS filtering. This will provide a range of readers with the expertise of implementing such solutions in FPGA hardware with a clear treatise of the mapping of algorithmic complexity into FPGA hardware which the authors believe is missing from current literature. The book summarizes over 30 years of learned experience.

A key focus of the book has been to look at the FPGA as a heterogeneous platform which can be used to construct complex DSP systems. In particular, we take a system-level approach, addressing issues such as system-level optimization, implementation and integration of IP cores, system communications frameworks and implementation for low power, to mention but a few. The

intention is that the designer will be able to apply some of the techniques developed in the book and use the examples along with existing C-based or HDL-based tools to develop solutions for their own specific application.

Organization

The purpose of the book is to give insights with examples of the challenges of implementing digital signal processing systems using FPGA technology; it does this by concentrating on the high-level mapping of DSP algorithms into suitable circuit architectures and not so much on the detailed FPGA specific optimizations as this. This topic is addressed more effectively in other texts and also increasingly, by HDL-based design tools. The focus of this text is to treat the FPGA as a hardware resource that can be used to create complex DSP systems. Thus the FPGA can be viewed as a heterogeneous platform comprising complex resources such as hard and soft processors, dedicated DSP blocks and processing elements connected by programmable and fast dedicated interconnect. The book is organized into four main sections.

The first section, effectively Chapters 2–5 covers the basics of both DSP systems and implementation technologies and thus provides an introduction to both of these areas. Chapter 2 starts with a brief treatise on DSP, covering both digital filtering and transforms. As well as covering basic filter structures, the text gives details on adaptive filtering algorithms. With regard to transforms, the chapter briefly covers the FFT, DCT and the discrete wavelet transform (DWT). Some applications in electrocardiogram (EEG) are given to illustrate some key points. This is not a detailed DSP text on the subject, but has been included to provide some background to the examples that are described later in the book.

Chapter 3 is dedicated to the computer arithmetic as it is an important topic for DSP system implementation. This starts with consideration of number systems and basic arithmetic functions, leading to adders and multipliers. These represent core blocks in FPGAs, but consideration is then given to circuits for performing square root and division as these are required in some DSP applications. A brief introduction is made to other number representations, namely signed digit number representations (SDNRs), logarithmic number systems (LNS), residue number systems (RNS) and coordinate rotation digital computer (CORDIC). However, this is not detailed as none of the examples use these number systems.

Chapter 4 covers the various technologies available to implement DSP algorithms. It is important to understand the other technology offerings so that the user can opt to choose the most suitable technology. Where possible, FPGA technology is compared with these other approaches with the differences clearly highlighted. Technologies covered include microprocessors with a focus on the ARM processor and DSPμs with detailed description given on the TMS320C64 series family from Texas Instruments. Parallel machines are then introduced, including systolic arrays, single instruction multiple data (SIMD) and multiple instruction multiple data (MIMD). Two examples of SIMD machines are then given, namely the Imagine processor and the Clearspeed processor.

In the final part of this first section, namely Chapter 5, a detailed description of commercial FPGAs is given, concentrating on the two main vendors, namely Xilinx and Altera, specifically their Virtex™ and Stratix® FPGA families, but also covering technology offerings from Lattice®, Atmel®, and Actel®. The chapter gives details of the architecture, DSP specific processing capability, memory organization, clock networks, interconnection frameworks and I/Os and external memory interfaces.

The second section of the book covers the system-level implementation in three main stages namely: efficient implementation from circuit architecture onto specific FPGA families; creation of circuit architecture from signal flow graph (SFG) representation and; system-level specification and implementation methodologies from a high-level model of computation representations. The first

chapter in this part, Chapter 6 covers the efficient implementation of FPGA designs from circuit architecture descriptions. As this has been published extensively, the chapter only gives a review of these existing techniques for efficient DSP implementation specifically distributed arithmetic (DA), but also the reduced coefficient multiplier (RCM) which has not been described in detail elsewhere. These latter techniques are particularly attractive for fixed coefficient functions such as fixed filters and transforms such as the DCT. The chapter also briefly discusses detailed design issues such as memory realization and implementation of delays.

Chapter 7 then gives an overview of the tools for performing rapid design and covers system specification in the form of Petri nets and other MoCs for high level embedded systems. Tools covered include Gedae, Compaan, ESPAM, Daedalus and Koski. The chapter also looks at IP core generation tools for FPGAs, including Labview FPGA and Synplify DSP as well as C-based rapid IP core design tools including MATLAB®.

The next stage of how DSP algorithms in the form of SFGs or dataflow graphs (DFGs) are mapped into circuit architectures which was the starting point for the technique described in Chapter 6, is then described in Chapter 8. This work is based on the excellent text by K. K. Parhi *VLSI Digital Signal Processing Systems : Design and Implementation*, Wiley, 1999, which describes how many of the techniques can be applied to VLSI-based signal processing systems. The chapter describes how DFG descriptions can be transformed for varying levels of parallelism and pipelining to create circuit architectures which best match the application requirements. The techniques are demonstrated using simple FIR and IIR filters.

Chapter 9 then presents the IRIS tool which has been developed to specifically capture the processes of creating circuit architecture from, in this case, SFG descriptions of DSP systems and algorithms involving many of the features described in Chapter 8. It demonstrates this for WDFs and specifically shows how hierarchy can be a major issue in system-level design, proposing the *white box* methodology as a possible approach. These chapters set the scene for the system-level issues described in the rest of the book.

The final stage of the book, namely Chapters 10 and 12 represents the third aspect of this design challenge, highlighting on high-level design. Chapters 8 and 9 have shown how to capture some level of DSP functionality to produce FPGA implementations. In many cases, these will represent part of the systems and could be seen as an efficient means of producing DSP IP cores. Chapter 10 gives some detailed consideration to the concept of creating silicon IP cores, highlighting the different flavours, namely *hard, soft and firm*, and illustrating the major focus for *design for reuse* which is seen as a key means of reducing the *design productivity gap*. Generation of IP cores has been a growth industry that has had a long association with FPGAs; indeed, attaining highly efficient FPGA solutions in a short design time has been vital in the use of FPGAs for DSP. Details of core generation based on real company experience is described in Chapter 10, along with a brief history of IP core evolution. The whole process of how parameterizable IP cores are created is then reviewed, along with a brief description of current FPGA IP core offerings from Xilinx and Altera.

Moving along with high-level design focus, Chapter 11 considers model-based design for heterogeneous FPGA. In particular, it focuses on dataflow modelling as a suitable platform for DSP systems and introduces the various flavours, including, synchronous dataflow, cyclo-static dataflow, multidimensional synchronous dataflow. Rapid synthesis and optimization techniques for creating efficient embedded software solutions from DFGs are then covered with topics such as graph balancing, clustering, code generation and DFG actor configurability. The chapter then outlines how it is possible to include pipelined IP cores via the *white box* concept using two examples, namely a normalized lattice filter (NLF) and a fixed beamformer example.

Chapter 12 then looks in detail at the creation of a soft, highly parameterizable core for RLS filtering. It starts with an introduction to adaptive beamforming and the identification of a QR-based algorithm as an efficient means to perform the beamforming. The text then clearly demonstrates how a series of architectures, leading to a single generic architecture, are then developed from the

algorithmic description. Issues such as a choice of fixed- and floating-point arithmetic and control overhead are also considered.

Chapter 13 then addresses a vital area for FPGA implementation and indeed, other forms of hardware, namely that of low power design. Whilst FPGAs are purported as a low power solution, this is only the case when compared with microprocessors, and there is quite a gap when FPGA implementations are compared with their ASIC counterparts. The chapter starts with a discussion on the various sources of power consumption, principally static and dynamic, and then presents a number of techniques to first reduce static power consumption which is limited due to the fixed nature of FPGA architecture and then dynamic power consumption which largely involves reducing the switched capacitance of the specific FPGA implementation. An FFT-based implementation is used to demonstrate some of the gains that can be achieved in reducing the consumed power.

Finally, Chapter 14 summarizes the main approaches covered in the text and considers some future evolutions in FPGA architectures that may be introduced. In addition, it briefly covers some topics not covered in the book, specifically reconfigurable systems. It is assumed that one of the advantages of FPGAs is that they can be programmed at start-up, allowing changes to be made to the design between operation cycles. However, considerable thought has been given to dynamically reconfiguring FPGAs, allowing them to be changed during operation, i.e. *dynamically* (where the previous mode can be thought of as *static* reconfiguration). This is interesting as it allows the FPGA to be viewed as virtual hardware, allowing the available hardware to implement functionality well beyond the capacity available on the current FPGA device. This has been a highly attractive proposition, but the practical realities somewhat limit its feasibility.

Acknowledgements

The authors have been fortunate to receive valuable help, support and suggestions from numerous colleagues, students and friends. The authors would like to thank Richard Walke and John Gray for motivating a lot of the work at Queen's University Belfast on FPGA. A number of other people have also acted to contribute in many other ways to either provide technical input or support. These include: Steve Trimberger, Ivo Bolsens, Satnam Singh, Steve Guccione, Bill Carter, Nabeel Shirazi, Wayne Luk, Peter Cheung, Paul McCambridge, Gordon Brebner and Alan Marshall.

The authors' research described in this book has been funded from a number of sources, including the Engineering and Physical Sciences Research Council, Ministry of Defence, Defence Technology Centre, Qinetiq, BAE Systems, Selex and Department of Education and Learning for Northern Ireland.

Several chapters are based on joint work that was carried out with the following colleagues and students, Richard Walke, Tim Harriss, Jasmine Lam, Bob Madahar, David Trainor, Jean-Paul Heron, Lok Kee Ting, Richard Turner, Tim Courtney, Stephen McKeown, Scott Fischaber, Eoin Malins, Jonathan Francey, Darren Reilly and Kevin Colgan.

The authors thank Simone Taylor and Nicky Skinner of John Wiley & Sons for their personal interest and help and motivation in preparing and assisting in the production of this work.

Finally the authors would like to acknowledge the support from friends and family including, Pauline, Rachel, Andrew, Beth, Anna, Lixin Ren, David, Gerry and the Outlaws, Colm and David.

1

Introduction to Field-programmable Gate Arrays

1.1 Introduction

Electronics revolutionized the 20th century and continues to make an impact in the 21st century. The birth and subsequent growth of the computer industry, the creation of mobile telephony and the general digitization of television and radio services has largely been responsible for the recent growth. In the 1970s and 1980s, electronic systems were created by aggregating standard components such as microprocessors and memory chips with digital logic components, e.g. dedicated integrated circuits (ICs) along with dedicated input/output (I/O) components on printed circuit boards (PCBs). As levels of integration grew, manufacturing working PCBs became more complex, largely due to increased component complexity in terms of the increase in the number of transistors and I/O pins but also the development of multi-layer boards with up to as many as 20 separate layers. Thus, the probability of incorrectly connecting components also grew, particularly as the possibility of successfully designing and testing a working system before production was coming under increasingly limited time pressure.

The problem was becoming more intense due to the difficulty that system descriptions were evolving as boards were being developed. Pressure to develop systems to meet evolving standards, or that could change after the board construction due to system alterations or changes in the design specification, meant that the concept of having a 'fully specified' design in terms of physical system construction and development on processor software code, was becoming increasingly unlikely. Whilst the use of programmable processors such as microcontrollers and microprocessors gave some freedom to the designer to make alterations in order to correct or modify the system after production, this was limited as changes to the interconnections of the components on the PCB, was only limited to I/O connectivity of the processors themselves. Thus the attraction of using programmability interconnection or 'glue logic' offered considerable potential and so the concept of field-programmable logic (FPL) specifically field-programmable gate array (FPGA) technology, was borne.

1.1.1 Field-programmable Gate Arrays

FPGAs emerged as simple 'glue logic' technology, providing programmable connectivity between major components where the programmability was based on either antifuse, EPROM or SRAM

FPGA-based Implementation of Signal Processing Systems R. Woods, J. McAllister, G. Lightbody and Y. Yi
© 2008 John Wiley & Sons, Ltd

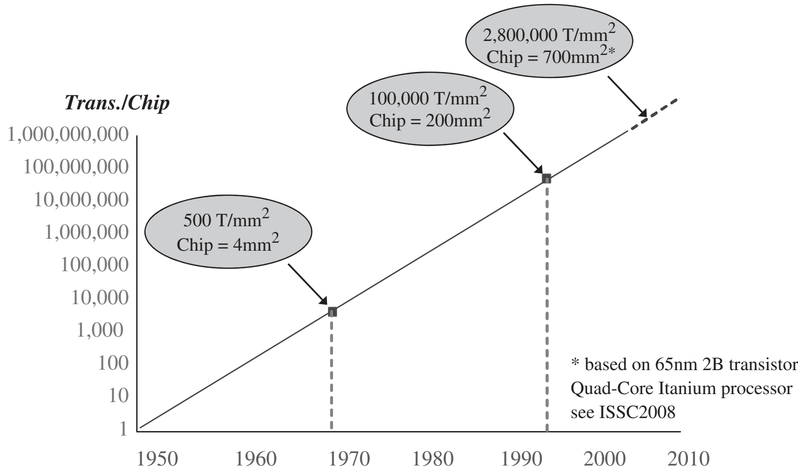


Figure 1.1 Moore's law (Moore 1965)

technologies (Maxfield 2004). This approach allows design errors which had only been recognized at this late stage of development to be corrected, possibly by simply reprogramming the FPGA thereby allowing the interconnectivity of the components to be changed as required. Whilst this approach introduced additional delays due to the programmable interconnect, it avoids a costly and time-consuming board redesign and considerably reduced the design risks.

Like many other electronics industries, the creation and growth in the market has been driven by Moore's law (Moore 1965), represented pictorially in Figure 1.1. Moore's law shows that the number of transistors has been doubling every 18 months. The incredible growth has led to the creation of a number of markets and is the driving force between the markets of many electronics products such as mobile telephony, digital musical products, digital TV to name but a few. This is because not only have the number of transistors doubled at this rate, but the costs have not increased, thereby reducing the cost per transistor at every technology advance. This has meant that the FPGA market has grown from nothing in just over 20 years to being a key player in the IC industry with a market judged to be of the order of US$ 4.0 billion.

On many occasions, the growth indicated by Moore's law has led people to argue that transistors are essentially free and therefore can be *exploited* as in the case of programmable hardware, to provide additional flexibility. This could be backed up by the observation that the cost of a transistor has dropped from one-tenth of a cent in the 1980s to one-thousandth of a cent in the 2000s. This observation could be argued to have been validated by the introduction of hardware programmability into electronics in the form of FPGAs. In order to make a single transistor programmable in an SRAM technology, the programmability is controlled by storing a '1' or a '0' on the gate of the transistor, thereby making it conduct or not. This value is then stored in an SRAM cell which typically requires six transistors, involving a 600% increase for the introduction of programmability. The reality is that in an overall FPGA implementation, the penalty is nowhere as harsh as this, but it has to be taken into consideration in terms of ultimate system cost.

It is the ability to program the FPGA hardware after fabrication that is the main appeal of the technology as it provides a new level of reassurance in an increasingly competitive market where 'right first time' system construction is becoming more difficult to achieve. It would appear that assessment was vindicated as in the late 1990s and early 2000s, when there was a major market downturn, the FPGA market remained fairly constant when other microelectronic technologies

were suffering. Of course, the importance of programmability has already been demonstrated by the microprocessor, but this represented a new change in how programmability was performed.

1.1.2 Programmability and DSP

The argument developed in the previous section presents a clear advantage of FPGA technology in terms of the use of its programmability to reduce the risk of incorrectly creating PCBs or evolving the manufactured product to later changes in standards. Whilst this might have been true in the early days of FPGA technology, evolution in silicon technology has moved the FPGA from being a programmable *interconnection* technology to making it into a system component. If the microprocessor or microcontroller was viewed as *programmable* system component, the current FPGA devices must also be viewed in this vein, giving us a different perspective on *system programmability*.

In electronic system design, the main attraction of microprocessors/microcontrollers is that it considerably lessens the risk of system development by reducing design complexity. As the hardware is fixed, all of the design effort can be concentrated on developing the *code* which will make the hardware work to the required system specification. This situation has been complemented by the development of efficient software compilers which have largely removed the need for designer to create assembly language; to some extent, this can absolve the designer from having a detailed knowledge of the microprocessor architecture (although many practitioners would argue that this is essential to produce *good* code). This concept has grown in popularity and embedded microprocessor courses are now essential parts of any electrical/electronic or computer engineering degree course.

A lot of this process has been down to the software developer's ability to exploit an underlying processor architecture, the Von Neumann architecture. However, this advantage has also been the limiting factor in its application to the topic of this text, namely digital signal processing (DSP). In the Von Neumann architecture, operations are processed sequentially, which allows relative straightforward interpretation of the hardware for programming purposes; however, this severely limits the performance in DSP applications which exhibit typically, high levels of parallelism and in which, the operations are highly data independent – allowing for optimisations to be applied. This cries out for parallel realization and whilst DSP microprocessors (here called DSPμs) go some way to addressing this situation by providing concurrency in the form of parallel hardware and software 'pipelining', there is still the concept of *one* architecture suiting *all* sizes of the DSP problem.

This limitation is overcome in FPGAs as they allow what can be considered to be a second level of programmability, namely programming of the underlying processor architecture. By creating an architecture that best meets the algorithmic requirements, high levels of performance in terms of area, speed and power can be achieved. This concept is not new as the idea of deriving a system architecture to suit algorithmic requirements has been the cornerstone of *application-specific integrated circuit* or ASIC implementations. In high volumes, ASIC implementations have resulted in the most cost effective, fastest and lowest energy solutions. However, increasing mask costs and impact of 'right first time' system realization have made the FPGA, a much more attractive alternative. In this sense, FPGAs capture the performance aspects offered by ASIC implementation, but with the advantage of programmability usually associated with programmable processors. Thus, FPGA solutions have emerged which currently offer several hundreds of gigaoperations per second (GOPS) on a single FPGA for some DSP applications which is at least an order of magnitude better performance than microprocessors.

Section 1.2 puts this evolution in perspective with the emergence of silicon technology by considering the history of the microchip. It highlights the key aspect of programmability which is

discussed in more detail in Section 1.3 and leads into the challenges of exploiting the advantages offered by FPGA technology in Section 1.4.

1.2 A Short History of the Microchip

Many would argue that the industrial revolution in the late 1700s and early 1800s had a major social impact on how we lived and travelled. There is a strong argument to suggest that the emergence of the semiconductor market has had a similar if not more far-reaching impact on our lives. Semiconductor technology has impacted how we interact with the world and each other through technologies such as mobile telephony, e-mail, videoconferencing, are entertained via TV, radio, digital video, are educated through the existence of computer-based learning, electronic books; and also how we work with remote working now possible through wireless communications and computer technology.

This all started with the first transistor that was discovered by John Bardeen and Walter Brattain whilst working for William Shockley in Bell Laboratories. They were working with the semiconductor material silicon, to investigate its properties, when they observed that controlling the voltage on the 'base' connector, would control the flow of electrons between the emitter and collector. This had a considerable impact for electronics, allowing the more reliable transistor to replace the vacuum tube and leading to a number of 'transistorized' products.

Another major evolution occurred in the development of the first silicon chip, invented independently by Jack Kilby and Robert Noyce, which showed it was possible to integrate components on a single block of semiconductor material hence the name *integrated circuit*. In addition, Noyce's solution resolved some practical issues, allowing the IC to be more easily mass produced. There were many advantages to incorporating transistor and other components onto a single chip from a manufacturing and design point-of-view. For example, there was no more need for separate components with manually assembled wires to connect them. The circuits could be made smaller and the manufacturing process could be automated. The evolution of the chip led to the development of the standard TTL 7400 series components pioneered by Texas Instruments and the building components of many basic electronics kits. It was not known at the time, but these chips would become a standard in themselves.

Another key innovation was the development of the first microprocessor, the Intel 4004 by Bob Noyce and Gordon Moore in 1968. It had just over 2300 transistors in an area of only 12 mm^2 which can be compared with today's 64-bit microprocessors which have 5.5 million transistors performing hundreds of millions of calculations each second. The key aspect was that by changing the programming code within the memory of the microprocessor, the function could be altered without the need to create a new hardware platform. This was fundamental to freeing engineers from the concept of building design by components which could not be easily changed to having a programmable platform where the functionality could be changed by altering the program code. It was later in 1965 in (Moore 1965) that Gordon Moore made the famous observation that has been coined as Moore's law. The original statement indicated that *the complexity for minimum component costs has increased at a rate of roughly a factor of two per year*, although this was later revised to every 18 months. This is representative of the evolution of silicon technology that allows us to use transistors, not only to provide functionality in processing data, but simply to create the overhead to allow us to provide programmability. Whilst this would suggest we could use transistors freely and that the microprocessor will dominate, the bottom line is that we are not using these transistors efficiently. There is an overall price to be paid for this in terms of the power consumed, thus affecting the overall performance of the system. In microprocessor systems, only a very small proportion of the transistors are performing useful work towards the computation.

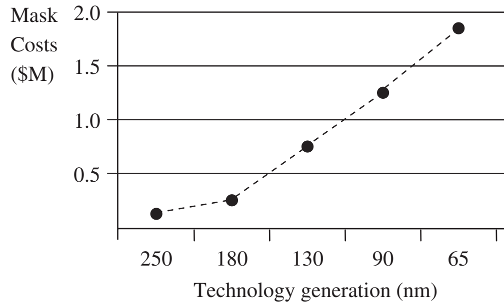


Figure 1.2 Mask cost versus technology generation (Zuchowski *et al*. 2002)

At this stage, a major shift in the design phase opened up the IC design process to a wide range of people, including university students (such as the main author at that time!). Mead and Conway (1979) produced a classic text which considerably simplified the IC design rules, allowing small chips to be implemented even without the need for design rule checking. By making some *worst case* assumptions, they were able to create a much smaller design rule set which could, given the size of the chips at that time, be performed manually. This lead to the 'demystifying' of the chip design process and with the development of software tools, companies were able to create ASICs for their own products. This along with the MOSIS program in the US (Pina 2001), provided a mechanism for IC design to be taught and experienced at undergraduate and postgraduate level in US universities. Later, the Eurochip program now known as Europractice (Europractice 2006) provided the same facility allowing a considerable number of chips to be fabricated and design throughout European universities. However, the ASIC concept was being *strangled* by increasing nonrecurrent engineering (NRE) costs which meant that there was an increased emphasis on 'right first time' design. These NRE costs were largely governed by the cost of generating masks for the fabrication process; these were increasing as it was becoming more expensive (and difficult) to generate the masks for finer geometries needed by shrinking silicon technology dimensions. This issue has become more pronounced as illustrated in the graph of Figure 1.2, first listed in Zuchowski *et al*. (2002) which gives the increasing cost (part of the NRE costs) needed to generate the masks for an ASIC.

The FPGA concept emerged in 1985 with the XC2064^{TM} FPGA family from Xilinx. At the same time, a company called Altera were also developing a programmable device, later to become EP1200 device which was the first high-density programmable logic device (PLD). Altera's technology was manufactured using 3-μm CMOS erasable programmable read-only-memory (EPROM) technology and required ultraviolet light to erase the programming whereas Xilinx's technology was based on conventional static RAM technology and required an EPROM to store the programming. The co-founder of Xilinx, Ross Freeman argued that with continuously improving silicon technology, transistors were going to increasingly get cheaper and could be used to offer programmability. This was the start of an FPGA market which was then populated by quite a number of vendors, including Xilinx, Altera, Actel, Lattice, Crosspoint, Algotronix, Prizm, Plessey, Toshiba, Motorola, and IBM. The market has now grown considerably and Gartner Dataquest indicated a market size growth to 4.5 billion in 2006, 5.2 billion in 2007 and 6.3 billion in 2008. There have been many changes in the market. This included a severe rationalization of technologies with many vendors such as Crosspoint, Algotronix, Prizm, Plessey, Toshiba, Motorola, and IBM disappearing from the market and a reduction in the number of FPGA families as well as the emergence of SRAM technology as the dominant technology largely due to cost. The market is now dominated by Xilinx

Table 1.1 Three ages of FPGAs

Period	Age	Comments
1984–1991	Invention	Technology is limited, FPGAs are much smaller than the application problem size Design automation is secondary Architecture efficiency is key
1992–1999	Expansion	FPGA size approaches the problem size Ease-of-design becomes critical
2000–2007	Accumulation	FPGAs are larger than the typical problem size Logic capacity limited by I/O bandwidth

and Altera and more importantly, the FPGA has grown from being a simple glue logic component to representing a complete System on Programmable Chip (SoPC) comprising on-board physical processors, soft processors, dedicated DSP hardware, memory and high-speed I/O.

In the 1990s, energy considerations became a key focus and whilst by this time, FPGAs had heralded the end of the gate array market, ASIC was still seen for the key mass market areas where really high performance and/or energy considerations were seen as key drivers such as mobile communications. Thus graphs comparing performance metrics for FPGA, ASIC and processor were generated and used by each vendor to indicate design choices. However, this is simplistic and a number of other technologies have emerged over the past decade and are described in Section 1.2.1.

The FPGA evolution was neatly described by Steve Trimberger given in his plenary talk (Trimberger 2007) and summarized in Table 1.1. It indicates three different eras of evolution of the FPGA. The age of *invention* where FPGAs started to emerge and were being used as system components typically to provide programmable interconnect giving protection to design evolutions and variations as highlighted in Section 1.1. At this stage, design tools were primitive, but designers were quite happy to extract the best performance by dealing with LUTs or single transistors. In the early 1990s, there was a rationalization of the technologies described in the earlier paragraphs and referred to by Trimberger as the *great architectural shakedown* where the technology was rationalized. The age of *expansion* is where the FPGA started to approach the problem size and thus design complexity was key. This meant that it was no longer sufficient for FPGA vendors to just produce place and route tools and so it was critical that HDL-based flows were created. The final evolution period is described as the period of *accumulation* where FPGA started to incorporate processors and high-speed interconnection. This is described in detail in Chapter 5 where the recent FPGA offerings are reviewed.

1.2.1 Technology Offerings

In addition to FPGAs, ASICs and microprocessors, a number of other technologies emerged over the past decade which are worth consideration. These include:

Reconfigurable DSP processors. These types of processors allow some form of customization whilst providing a underlying fixed type of architecture that provides some level of functionality for the application required. Examples include the Xtensa processor family from Tensilica (Tensilica Inc. 2005) and D-Fabrix from Elixent (now Panasonic) which is a reconfigurable semiconductor intellectual property (IP) (Elixent 2005)

Structure ASIC implementation It could be argued that the concept of 'gate array' technology has risen again in the form of *structured ASIC* which is a predefined silicon framework where the

user provides the interconnectivity in the form of reduced silicon fabrication. This option is also offered by Altera through their Hardcopy technology (Altera Corp. 2005), allowing users to migrate their FPGA design direct to ASIC.

The current situation is that quite a number of these technologies that now co-exist are targeted at different markets. This section has highlighted how improvements in silicon technologies have seen the development of new technologies which now form the electronic hardware for developing systems, in our case, DSP systems.

A more interesting viewpoint is to consider the availability of programmability in these technologies. The mask cost issue highlighted in Figure 1.2, along with the increasing cost of fabrication facilities, paints a depressing picture for developing application-specific solutions. This would tend to suggest that dedicated silicon solutions will be limited to mass market products and will only be able to exploited by big companies who can take the risk. Nanotechnology is purported to be a solution, but this will not be viable within the next decade in the authors' opinion. Structured ASIC could be viewed as a re-emergence of the gate array technology (at least in terms of the concept of constructing the technology) and will provide an interesting solution for low-power applications. However, the authors would argue that the availability of programmability will be central to next generation systems where time-to-market, production costs and pressures of right-first-time hardware are becoming so great that the concept of being able to program hardware will be vital. The next section attempts to compare technologies with respect to programmability.

1.3 Influence of Programmability

In many texts, Moore's law is used to highlight the evolution of silicon technology. Another interesting viewpoint particularly relevant for FPGA technology, is *Makimoto's wave* which was first published in the January 1991 edition of *Electronics Weekly*. It is based on an observation by Tsugio Makimoto who noted that technology has shifted between standardization and customization (see Figure 1.3). In the early 1960s, a number of standard components were developed, namely the Texas Instruments 7400 TTL series logic chips and used to create applications. In the early 1970s, the custom LSI era was developed where chips were created (or customized) for specific applications such as the calculator. The chips were now increasing in their levels of integration

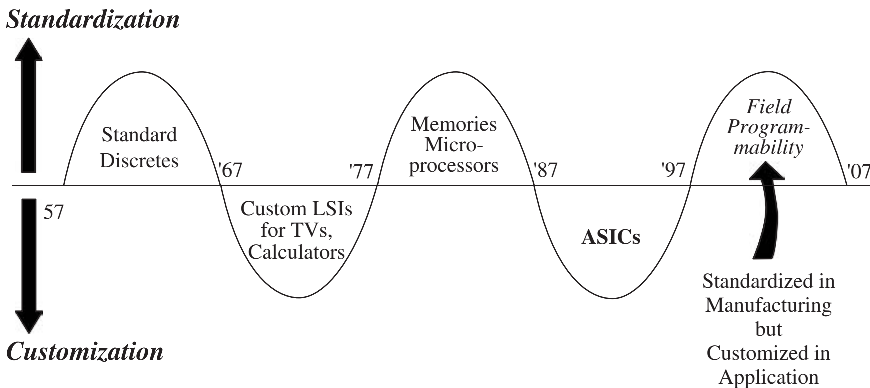


Figure 1.3 Makimoto's wave. Reproduced by permission of Reed Business Information

and so the term *medium-scale integration* (MSI) was born. The evolution of the microprocessor in the 1970s saw the swing back towards standardization where one 'standard' chip was used for a wide range of applications. The 1980s then saw the birth of ASICs where designers could overcome the limitations of the sequential microprocessor which posed severe limitations in DSP applications where higher levels of computations were needed. The DSP processor also emerged, such as the TMS32010, which differed from conventional processors as they were based on the *Harvard* architecture which had separate program and data memories and separate buses. Even with DSP processors, ASICs offered considerable potential in terms of processing power and more importantly, power consumption. The emergence of the FPGA from a 'glue component' that allows other components to be connected together to form a system to becoming a system component or even a system itself, led to increased popularity. The concept of coupling microprocessors with FPGAs in heterogeneous platforms was considerably attractive as this represented a completely programmable platform with microprocessors to implement the control-dominated aspects of DSP systems and FPGAs to implement the data-dominated aspects. This concept formed the basis of FPGA-based custom computing machines (FCCMs) which has led to the development of several conferences in the area and formed the basis for 'configurable' or *reconfigurable computing* (Villasenor and Mangione-Smith 1997). In these systems, users could not only implement computational complex algorithms in hardware, but use the programmability aspect of the hardware to change the system functionality allowing the concept of 'virtual hardware' where hardware could 'virtually' implement systems, an order of magnitude larger (Brebner 1997). The concept of reconfigurable systems is reviewed in Chapter 14.

We would argue that there have been two programmability eras with the first era occurring with the emergence of the microprocessor in the 1970s, where engineers could now develop programmable solutions based on this fixed hardware. The major challenge at this time was the software environments; developers worked with assembly language and even when compilers and assemblers emerged for C, best performance was achieved by hand coding. Libraries started to appear which provided basic common I/O functions, thereby allowing designers to concentrate on the application. These functions are now readily available as core components in commercial compiler and assemblers. Increasing the need for high-level languages grew and now most programming is carried out in high-level programming languages such as C and Java with an increased use of even higher level environments such as UML.

The second era of programmability is offered by FPGAs. In the diagram, Makimoto indicates that the field programmability is standardized in manufacture and customized in application. This can be considered to have offered hardware programmability if you think in terms of the first wave as the programmability in the software domain where the hardware remains fixed. This is a key challenge as most of computer programming tools work on the principle of fixed hardware platform, allowing optimizations to be created as there is a clear direction on how to improve performance from an algorithmic representation. With FPGAs, the user is given full freedom to define the architecture which best suits the application. However, this presents a problem in that each solution must be *handcrafted* and every hardware designer knows the issues in designing and verifying hardware designs!

Some of the trends in the two eras have similarities. In the earlier days, schematic capture was used to design early circuits which was synonymous with assembly level programming. Hardware description languages such as VHDL and Verilog then started to emerge that could used to produce a higher level of abstraction with the current aim to have C-based tools such as SystemC and CatapultC from Mentor Graphics as a single software based programming environment. Initially as with software programming languages, there was a mistrust in the quality of the resulting code produced by these approaches. However with the establishment of improved cost-effective, synthesis tools which was equivalent to evolution of efficient software compilers for high-level programming languages, and also the evolution of library functions, a high degree of confidence

was subsequently established and use of HDLs is now commonplace for FPGA implementation. Indeed, the emergence of IP cores mirrored the evolution of libraries such as I/O programming functions for software flows where common functions were reused as developers trusted the quality of the resulting implementation produced by such libraries, particularly as pressures to produce more code within the same time-span grew with evolving technology. The early IP cores emerged from basic function libraries into complex signal processing and communications functions such as those available from the FPGA vendors and various IP web-based repositories.

1.4 Challenges of FPGAs

In the early days, FPGAs were seen as glue logic chips used to plug components together to form complex systems. FPGAs then increasingly came to be seen as complete systems in themselves, as illustrated in Table 1.1. In addition to technology evolution, a number of other considerations accelerated this. For example, the emergence of the FPGA as a DSP platform was accelerated by the application of distributed arithmetic (DA) techniques (Goslin 1995, Meyer-Baese 2001). DA allowed efficient FPGA implementations to be realized using the LUT-based/adder constructs of FPGA blocks and allowed considerable performance gains to be gleaned for some DSP transforms such as fixed coefficient filtering and transform functions such as fast Fourier transform (FFT). Whilst these techniques demonstrated that FPGAs could produce highly effective solutions for DSP applications, the concept of squeezing the last aspect of performance out of the FPGA hardware and more importantly, spending several person months to create such innovative designs, was now becoming unacceptable. The increase in complexity due to technology evolution, meant that there was a growing gap in the scope offered by current FPGA technology and the designer's ability to develop solutions efficiently using currently available tools. This was similar to the 'design productivity gap' (IRTS 1999) identified in the ASIC industry where it was viewed that ASIC design capability was only growing at 25% whereas Moore's law growth was 60%. The problem is not as severe in FPGA implementation as the designer does not have to deal with sub-micrometre design issues. However, a number of key issues exist and include:

Understanding how to map DSP functionality into FPGA. Some of the aspects are relatively basic in this arena, such as multiplications, additions and delays being mapped onto on-board multipliers, adder and registers and RAM components respectively. However, the understanding of floating-point versus fixed-point, word length optimization, algorithmic transformation cost functions for FPGA and impact of routing delay are issues that must be considered at a system level and can be much harder to deal with at this level.

Design languages. Currently hardware description languages such as VHDL and Verilog and their respective synthesis flows are well established. However, users are now looking at FPGAs with the recent increase in complexity resulting in the integration of both fixed and programmable microprocessors cores as a complete system, and looking for design representations that more clearly represent system description. Hence there is an increased EDA focus on using C as a design language, but other representations also exist such as those methods based on models of computations (MoCs) such as synchronous dataflow.

Development and use of IP cores. With the absence of quick and reliable solutions to the design language and synthesis issues, the IP market in SoC implementation has emerged to fill the gap and allow rapid prototyping of hardware. *Soft cores* are particularly attractive as design functionality can be captured using HDLs and efficiently translated into the FPGA technology of choice in a highly efficient manner by conventional synthesis tools. In addition, processor cores have been developed which allow dedicated functionality to be added. The attraction of

these approaches are that they allow application specific functionality to be quickly created as the platform is largely fixed.

Design flow. Most of the design flow capability is based around developing FPGA functionality from some form of higher-level description, mostly for complex functions. The reality now is that FPGA technology is evolving at such a rate that systems comprising FPGAs and processors are starting to emerge as a SoC platform or indeed, FPGAs as a single SoC platform as they have on-board hard and soft processors, high-speed communications and programmable resource, and this can be viewed as a complete system. Conventionally, software flows have been more advanced for processors and even multiple processors as the architecture is fixed. Whilst tools have developed for hardware platforms such as FPGAs, there is a definite need for software for flows for heterogeneous platforms, i.e. those that involve both processors and FPGAs.

These represent the challenges that this book aims to address and provide the main focus for the work that is presented.

References

Altera Corp. (2005) Hardcopy structured asics: Asic gain without the paint. Web publication downloadable from http://www.altera.com.

Brebner G (1997) The swappable logic unit. *Proc. IEEE Symp. on FPGA-based Custom Computing Machines*, Napa, USA, pp. 77–86.

Elixent (2005) Reconfigurable algorithm processing (rap) technology. Web publication downloadable from http://www.elixent.com/.

Europractice (2006) Europractice activity report. Web publication downloadable from http://europractice-ic.com/documents_annual_reports.php.

Goslin G (1995) Using xilinx FPGAs to design custom digital signal processing devices. *Proc. DSPX*, pp. 565–604.

IRTS (1999) *International Technology Roadmap for Semiconductors*, 1999 edn. Semiconductor Industry Association. http://public.itrs.net

Maxfield C (2004) *The Design Warrior's Guide to FPGAs*. Newnes, Burlington.

Mead C and Conway L (1979) *Introduction to VLSI Systems*. Addison-Wesley Longman, Boston.

Meyer-Baese U (2001) *Digital Signal Processing with Field Programmable Gate Arrays*. Springer, Germany.

Moore GE (1965) Cramming more components onto integrated circuits. *Electronics*. Web publication downloadable from ftp://download.intel.com/research/silicon/moorespaper.pdf.

Pina CA (2001) Mosis: IC prototyping and low volume production service *Proc. Int. Conf. on Microelectronic Systems Education*, pp. 4–5.

Tensilica Inc. (2005) The Xtensa 6 processor for soc design. Web publication downloadable from http://www.tensilica.com/.

Trimberger S (2007) FPGA futures: Trends, challenges and roadmap *IEEE Int. Conf. on Field Programmable Logic*.

Villasenor J and Mangione-Smith WH (1997) Configurable computing. *Scientific American*, pp. 54–59.

Zuchowski P, Reynolds C, Grupp R, Davis S, Cremen B and Troxel B (2002) A hybrid ASIC and FPGA architecture. *IEEE/ACM Int. Conf. on Computer Aided Design*, pp. 187–194.

2

DSP Fundamentals

2.1 Introduction

In the early days of electronics, signals were processed and transmitted in their natural form, typically an analogue signal created from a source signal such as speech, then converted to electrical signals before being transmitted across a suitable transmission media such as a broadband connection. The appeal of processing signals digitally was recognized quite some time ago for a number of reasons. Digital hardware is generally superior and more reliable than its analogue counterpart which can be prone to ageing and can give uncertain performance in production. DSP on the other hand gives a guaranteed accuracy and essentially perfect reproducibility (Rabiner and Gold 1975). In addition, there is considerable interest in merging the multiple networks that transmit these signals, such as the telephone transmission networks, terrestrial TV networks and computer networks, into a single or multiple digital transmission media. This provides a strong motivation to convert a wide range of information formats into their digital formats.

Microprocessors, DSP micros and FPGAs perform a suitable platform for processing such digital signals, but it is vital to understand a number of basic issues with implementing DSP algorithms on, in this case, FPGA platforms. These issues range from understanding both the sampling rates and computational rates of different applications with the aim of understanding how these requirements affect the final FPGA implementation, right through to the number representation chosen for the specific FPGA platform and how these decisions impact the performance of the DSP systems. The choice of algorithm and arithmetic requirements can have severe implications on the quality of the final implementation.

The purpose of this chapter is to provide background and some explanation for many of these issues. It starts with a introduction to basic DSP concepts that affect hardware implementation, such as sampling rate, computational rate and latency. A brief description of common DSP algorithms is then given, starting with a review of transforms, including the fast Fourier transform (FFT), discrete cosine transform (DCT) and the discrete wavelet transform (DWT). The chapter then moves onto to review filtering and gives a brief description of finite impulse response (FIR), filters and infinite impulse response (IIR) filters and wave digital filters (WDFs). The final section on DSP systems is dedicated to adaptive filters and covers both the least-mean-squares (LMS) and recursive least-squares (RLS) algorithms. The final chapter of the book discusses the arithmetic implications of implementing DSP algorithms as the digitization of signals implies that the representation and processing of the signals are vital to the fidelity of the final system.

As the main aim of the book is in the implementation of such systems in FPGA hardware, the chapter aims to give the reader an introduction to DSP algorithms to such a level as to provide

FPGA-based Implementation of Signal Processing Systems R. Woods, J. McAllister, G. Lightbody and Y. Yi
© 2008 John Wiley & Sons, Ltd

grounding to many of the examples that are described later. A number of good introductory texts that explain the background of DSP systems can be found in the literature, ranging from the basic principles (Lynn and Fuerst 1994, Williams 1986) to more comprehensive texts (Rabiner and Gold 1975). Omondi's book on computer arithmetic is also recommended for an excellent text on computer arithmetic for beginners (Omondi 1994).

The chapter is organized as follows. Section 2.2 gives details on how signals are digitized and Section 2.3 describes the basic DSP concepts, specifically sampling rate, latency and pipelining that are relevant issues in FPGA implementation. Section 2.4 introduces DSP transforms and covers the fast Fourier transform (FFT), discrete cosine transform (DCT) and the discrete wavelet transform (DWT). Basic filtering operations are covered in Section 2.5 and extended to adaptive filtering in section 2.6.

2.2 DSP System Basics

There is an increasing need to process, interpret and comprehend information, including numerous industrial, military, and consumer applications. Many of these involve speech, music, images or video, or may support communication systems through error detection and correction, and cryptography algorithms. This involves real-time processing of a considerable amount of different types of content at a series of sampling rates ranging from single Hz as in biomedical applications, right up to tens of MHz as in image processing applications. In a lot of cases, the aim is to process the data to enhance part of the signal, such as edge detection in image processing or eliminating interference such as jamming signals in radar applications, or removing erroneous input, as in the case of echo or noise cancellation in telephony. Other DSP algorithms are essential in capturing, storing and transmitting data, audio, images and video; compression techniques have been used successfully in digital broadcasting and telecommunications.

Over the years, a lot of the need for such processing has been standardized, as illustrated by Figure 2.1 which gives an illustration of the algorithms required in a range of applications. In communications, the need to provide efficient transmission using orthogonal frequency division multiplexing (OFDM) has emphasized the need for circuits for performing the FFT. In image compression, the evolution initially of the joint photographic experts group (JPEG) and then the motion picture experts group (MPEG), led to the development of the JPEG and MPEG standards respectively; these standards involve a number of core DSP algorithms, specifically DCT and motion estimation and compensation.

The appeal of processing signals digitally was recognized quite some time ago as digital hardware is generally superior and more reliable than its analogue counterpart; analogue hardware can be prone to ageing and can give uncertain performance in production. DSP on the other hand, gives a guaranteed accuracy and essentially perfect reproducibility (Rabiner and Gold 1975). The main proliferation of DSP has been driven by the availability of increasingly cheap hardware, allowing the technology to be easily interfaced to computer technology, and in many cases, to be implemented on the same computers. The need for many of the applications mentioned in Figure 2.1 has driven the need for increasingly complex DSP systems which in turn has seen the growth of the research area involved in developing efficient implementation of some DSP algorithms. This has also driven the need for DSP micros covered in Chapter 3.

2.3 DSP System Definitions

The basic realisation of DSP systems given in Figure 2.2, shows how a signal is digitized using an analogue-to-digital (A/D) converter, processed in a DSP system before being converted back to an analogue signal. The digitised signal is obtained as shown in Figure 2.3 where an analogue signal

is converted into a pulse of signals and then quantized to a series of numbers. The input stream of numbers in digital format to the DSP system is typically labelled $x(n)$ and the output is given as $y(n)$. The original analogue signal can be derived from a range of source such as voice, music, medical or radio signal, a radar pulse or an image. Obviously, the representation of the data is a key aspect and this is considered in the next chapter. A wide range of signal processing can be carried out, as illustrated in Figure 2.1, as digitizing the signal opens up a wide domain of possibilities as to how the data can be manipulated, stored or transmitted.

A number of different DSP functions can be carried out either in the time domain, such as filtering, or operations in the frequency domain by performing an FFT (Rabiner and Gold 1975). The DCT forms the central mechanism for JPEG image compression which is also the foundation for the MPEG standards. This algorithm enables the components within the image that are invisible to the naked eye to be identified by converting the spatial image into the frequency domain. They can then be removed using quantization in the MPEG standard without a discernible degradation in

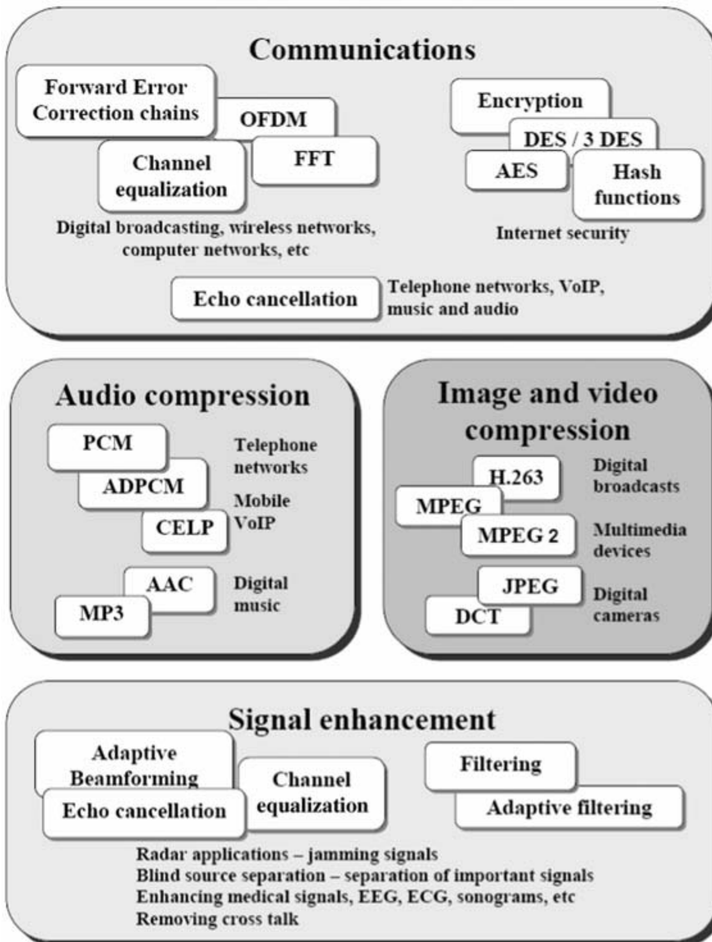


Figure 2.1 Example applications for DSP

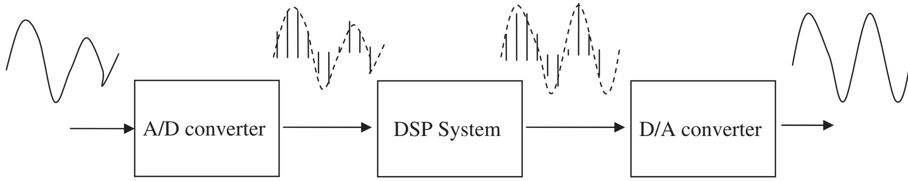


Figure 2.2 Basic DSP system

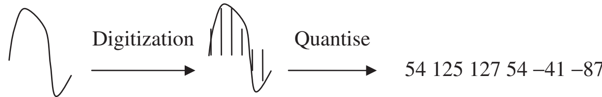


Figure 2.3 Digitization of analogue signals

the overall image quality. By increasing the amount of data removed, greater reduction in file size is achievable at a cost in image quality. Wavelet transforms offer both time domain and frequency domain information and have roles, not only in applications for image compression, but also for extraction of key information from signals and for noise cancellation. One such example is in extracting key features from medical signals such as the EEG.

2.3.1 Sampling Rate

An introduction to DSP systems would be incomplete without a brief reminder of the Nyquist–Shannon sampling theorem which states that the *exact reconstruction of a continuous-time baseband signal from its samples is possible if the signal is band limited and the sampling frequency is greater than twice the signal bandwidth*. For a more detailed explanation refer to the papers by Shannon and Nyquist (Nyquist 2002, Shannon 1949) and also some of the texts listed earlier in this chapter (Lynn and Fuerst 1994, Rabiner and Gold 1975, Williams 1986). In simple terms when digitizing an analogue signal the rate of sampling must be at least twice the maximum frequency f_m (within the signal being digitized) so as to maintain the information and prevent aliasing (Shannon 1949). In other words, the signal needs to be band limited, meaning that there is no spectral energy above a certain maximum frequency f_m. The Nyquist sampling rate f_s is then determined as $2f_m$.

A simple example is the sampling of the speech which is standardized at 8 kHz. This sampling rate is sufficient to provide an accurate representation of the spectral components of the speech signal as the spectral energy above 4 kHz and probably 3 kHz, does not contribute greatly to signal quality. In contrast, digitizing of music typically requires a sample rate for example of 44.2 kHz to cover the spectral range of 22.1 kHz as it is acknowledged that this is more than sufficient to cope with the hearing range of the human ear which typically cannot detect signals above 18 kHz. Moreover, this increase is natural due to the more complex spectral composition of music when compared with speech.

In other applications, the determination of the sampling rate does not just come down to human perception, but involves other aspects. Take, for example, the digitizing of medical signals such as the electroencephalogram (EEG) which are the result of electrical activity within the brain picked up from electrodes in contact with the surface of the skin. Through the means of capturing the information, the underlying waveforms can be heavily contaminated by noise. One particular application is a hearing test whereby a stimulus is applied to the subject's ear and the resulting EEG signal is observed at a certain location on the scalp. This test is referred to as the auditory

brainstem response (ABR) as it looks for an *evoked response* from the EEG in the brainstem region of the brain, within 10 ms from the stimulus onset. The ABR waveform of interest has a frequency range of 100–3000 Hz, therefore bandpass filtering of the EEG signal to this region is performed during the recording process prior to digitization. However, there is a slow response roll-off at the boundaries and unwanted frequencies may still be present. Once digitized the EEG signal may be filtered again, possibly using wavelet denoising to remove the upper and lower contaminating frequencies. The duration of the ABR waveform of interest is 20 ms, 10 ms prior to stimulus and 10 ms afterwards. The EEG is sampled at 20 kHz, therefore with a Nyquist frequency of 10 kHz, which exceeds twice the highest frequency component (3 kHz) present in the signal. This equates to 200 samples, before and after the stimulus.

Sampling rate is directly related to DSP computation rates and therefore, performance requirements in terms of the throughput rates required. Typical values are given in Figure 2.4, although sampling rate alone should not be used as the guide for technology choice. For example, take a 128 tap FIR filter for an audio application, the sampling rate may be 44.2 kHz, but the throughput rate required will be 11.2 Msample/s as during each sample as 128 multiplications and 127 additions (255 operations) need to be performed, at the sampling rate! If sampling rate is used as a misleading comparison, then the clock rate of the processor can also be viewed as a misleading metric. Clock rate is usually quoted by technology providers as a measure of possible performance, as throughput rate can then be computed by dividing the clock rate by the number of cycles needed to be performed per sample. Most noticeably, personal computers (PCs) quote clock rates as a metric of performance but as Chapter 3 demonstrates, it is the throughput rate that is more important.

2.3.2 Latency and Pipelining

Latency is the time required to produce a result after the input is fed into the system. In a synchronous system, this can be defined as the number of clock cycles which must evolve before the output is produced. For a simple system with inputs a_0 and x_n which produces an output y_n, the latency is as indicated in Figure 2.5 given as T_{Latency}, with the throughput rate given as $1/T_{\text{D}}$, where T_{D} is the time between successive outputs, and therefore inputs.

In synchronous systems, throughput is related to clock rate. Given that T_{D} is the critical path between registers within a digital circuit, this will determine the maximum achievable clock rate and will be directly related to the amount of logic and the physical distance between two registers. By adding additional pipelining stages the critical path within the circuit can be reduced, however, this is at a cost of increased latency and resulting area and power. Figure 2.6 illustrates a simple example of pipelining. Here, a computational block has six distinct stages. The diagram shows how by placing pipeline stages at certain locations within the circuit, the physical distance between registers can be reduced and hence the clock rate can be increased. The first example in Figure 2.6(a) has a latency of just one clock cycle, but can only be clocked at maximum of 36.4 MHz. Figure 2.6(b) shows pipelining cuts after each major logic block, resulting in a maximum clock rate of 100 MHz and a latency of six clock cycles. By moving one of the pipeline cuts to within one of the larger computational blocks (block C) while removing a pipeline cut after block D as shown in Figure 2.6(c), the critical path can be reduced to 5 ns, resulting in a clock

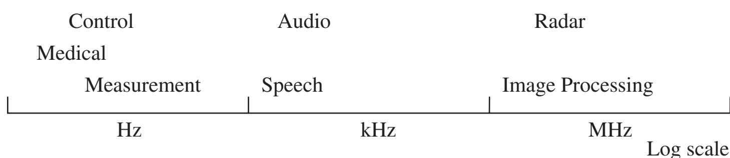


Figure 2.4 Sampling rates for many DSP systems

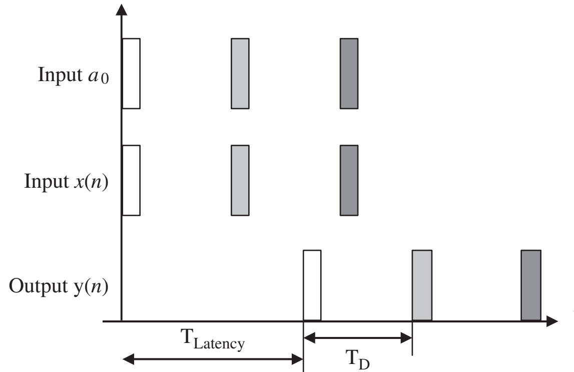


Figure 2.5 Illustration of latency for a simple DSP system

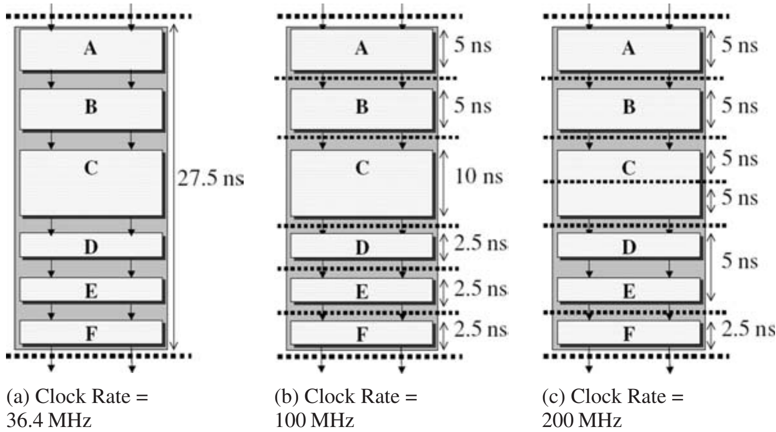


Figure 2.6 Simple pipelining example

rate of 200 MHz while maintaining a latency of 6 cycles. These figures demonstrate some basic theory behind circuit retiming, as important feature in efficient design practices and tools and will be covered in more detail in chapters 8 and 9.

2.4 DSP Transforms

This section will give a brief overview of some of the key DSP transforms mentioned in Section 2.2, including a brief description of applications.

2.4.1 Fast Fourier Transform

The Fourier transform is the transform of a signal from the time domain representation to the frequency domain representation. In basic terms it breaks a signal up into its constituent frequency components, representing a signal as a sum of a series of sines and cosines. The Fourier series

expansion of a periodic function, $f(t)$, is given in Equation (2.1) below

$$f(t) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty}[a_n cos(\omega_n t) + b_n sin(\omega_n t)] \tag{2.1}$$

where, for any non-negative integer n, ω_n is the nth harmonic in radians of $f(t)$ and is given below:

$$\omega_n = n\frac{2\pi}{T} \tag{2.2}$$

and a_n are the even Fourier coefficients of $f(t)$, given as

$$a_n = \frac{2}{T}\int_{t_1}^{t_2} cos(\omega_n t)\mathrm{d}t \tag{2.3}$$

and b_n are the odd Fourier coefficients, given as

$$b_n = \frac{2}{T}\int_{t_1}^{t_2} sin(\omega_n t)\mathrm{d}t \tag{2.4}$$

The Discrete Fourier transform (DFT), as the name suggests, is the discrete version of the continuous Fourier transform. applied to sampled signals. The input sequence is finite in duration and hence the DFT only evaluates the frequency components present in this portion of the signal. The inverse DFT will therefore only reconstruct using these frequencies and may not provide a complete reconstruction of the original signal (unless this signal is periodic). Equation (2.5) gives a definition for the DFT where the input sampled signal is represented by a sequence of complex numbers, $x(0), x(1), \ldots, x(N-1)$, and the transformed output is given by the sequence of complex numbers, $X(0), X(1), \ldots, X(N-1)$.

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-\frac{2\pi i}{N}kn} \tag{2.5}$$

where $k = 0, 1, \ldots, N-1$.

The fast Fourier transform (FFT) is a computationally efficient method for calculating the DFT. It has immense impact in a range of applications. One particular use is in the central computation within OFDM. This spread spectrum digital modulation scheme is used in communication, particularly within wireless technology, and has resulted in vastly improved data rates within the 802.11 standards, to name just one example. Here, the algorithm relies on the orthogonal nature of the frequency components extracted through the FFT, allowing each of these components to act as a subcarrier. Note that the receiver uses the inverse FFT (IFFT) to detect the subcarriers and reconstruct the transmission. The individual subcarriers are modulated using a typical low symbol rate modulation scheme such as phase-shift or quadrature amplitude modulation (QAM), depending on the application. For IEEE 802.11a, the data rate ranges up to 54 MBps depending on the environmental conditions and noise, i.e. phase shift modulation is used for the lower data rates when greater noise is present, QAM is used in less noisy environments reaching up to 54 MBps. Figure 2.7 gives an example of the main components within a typical communications chain.

The IEEE 802.11a wireless LAN standard using OFDM in the 5 GHz region of the US ISM band over a channel bandwidth of 125 MHz. From this bandwidth 52 frequencies are used, 48 for data and 4 for synchronization. The latter point is very important, as the basis on which OFDM works, i.e. orthogonality, relies on the receiver and transmitter being perfectly synchronized.

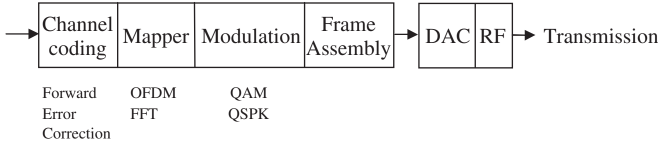


Figure 2.7 Wireless communications transmitter

2.4.2 Discrete Cosine Transform (DCT)

The DCT is based on the DFT, but uses only real numbers, i.e. the cosine part of the transform, as defined in Equation (2.6).

$$X(k) = \sum_{n=0}^{N-1} cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \tag{2.6}$$

where $k = 0, 1, \ldots, N-1$.

Its two-dimensional (2D) form given in Equation (2.7), is a vital computational component in the JPEG image compression and also features in MPEG standards.

$$F_{u,v} = \alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} f_{x,y} cos\left[\frac{\pi}{8}\left(x+\frac{1}{2}\right)u\right] cos\left[\frac{\pi}{8}\left(y+\frac{1}{2}\right)v\right] \tag{2.7}$$

where u is the horizontal spatial frequency for $0 \le u < 8$, v is the vertical spatial frequency for $0 \le v < 8$, $\alpha(u)$ and $\alpha(v)$ are constants, $f_{x,y}$ is the value of the (x, y) pixel and $F_{u,v}$ is the value of the (u, v) DCT coefficient. Figure 2.8 gives an illustration of the DCT applied to JPEG image compression. The DCT is performed on the rows and the columns of the image block of 8×8 pixels. The resulting frequency decomposition places the more important lower-frequency components at the top left-hand corner of the matrix, and the frequency of the components increase when moving towards the bottom right-hand part of the matrix. Once the image has been transformed into numerical values representing the frequency components, the higher-frequency components may be removed through the process of quantization as they will have less importance in image quality. Naturally, the greater the amount to be removed the higher the compression ratio; at a certain point the image quality will begin to deteriorate. This is referred to as lossy compression. The numerical values for the image are read in a zigzag fashion, as depicted in Figure 2.8.

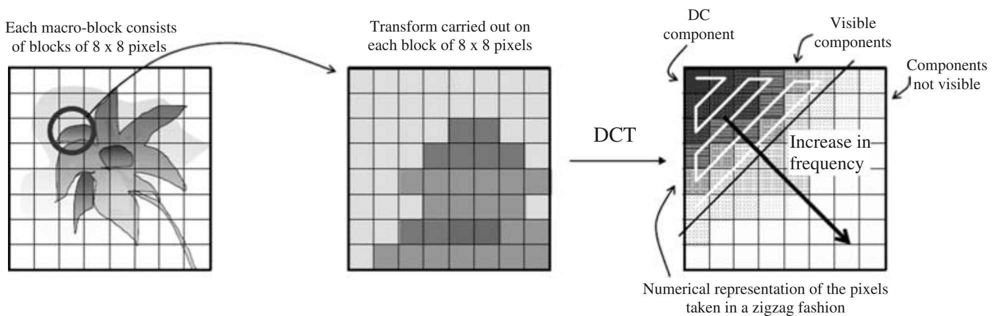


Figure 2.8 DCT applied to image compression

2.4.3 Wavelet Transform

A wavelet is a fast decaying waveform containing oscillations. Wavelet decomposition forms a powerful tool for multi-resolution filtering and analysis and is performed by correlating scaled versions of this original wavelet function (i.e. mother wavelet) against the input signal. This decomposes the signal into frequency bands that maintain a level of temporal information (Mallat 1989). This is particularly useful for frequency analysis of waveforms that are pseudo-stationary where the time-invariant FFT may not provide the complete information.

There are many families of wavelet equations such as the Daubechies, Coiflet and Symmlet (Daubechies 1992). Wavelet decomposition may be performed in a number of ways, namely the continuous wavelet transform (CWT) or the discrete wavelet transform (DWT) which is described in the next section.

2.4.4 Discrete Wavelet Transform

The DWT is performed using a series of filters, as illustrated in Figure 2.9 which shows a six-level wavelet decomposition. At each stage of the DWT, the input signal is passed though a high-pass and a low-pass filter, resulting in the *detail* and *approximation* coefficients such as those illustrated.

Equation (2.8) gives the equation for the low-pass filter. By removing half the frequencies at each stage, the signal information can be represented using half the number of coefficients, hence the equations for the low and high filters become Equations (2.9) and (2.10) respectively, where n has now become $2n$, representing the down-sampling process.

$$y(n) = (xg)(n) = \sum_{\infty}^{-\infty} x(k)g(n-k) \tag{2.8}$$

$$y_{\text{low}}(n) = \sum_{\infty}^{-\infty} x(k)g(2n-k) \tag{2.9}$$

$$y_{\text{high}}(n) = \sum_{\infty}^{-\infty} x(k)h(2n-k) \tag{2.10}$$

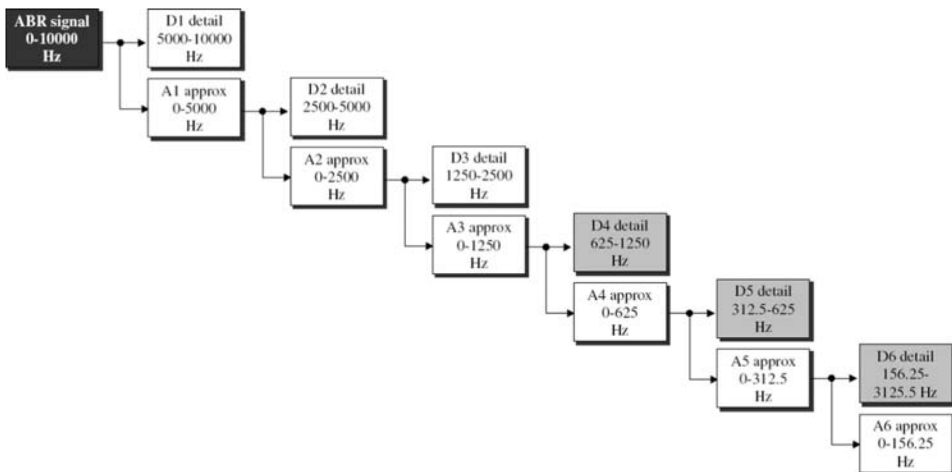


Figure 2.9 Discrete wavelet decomposition to six levels

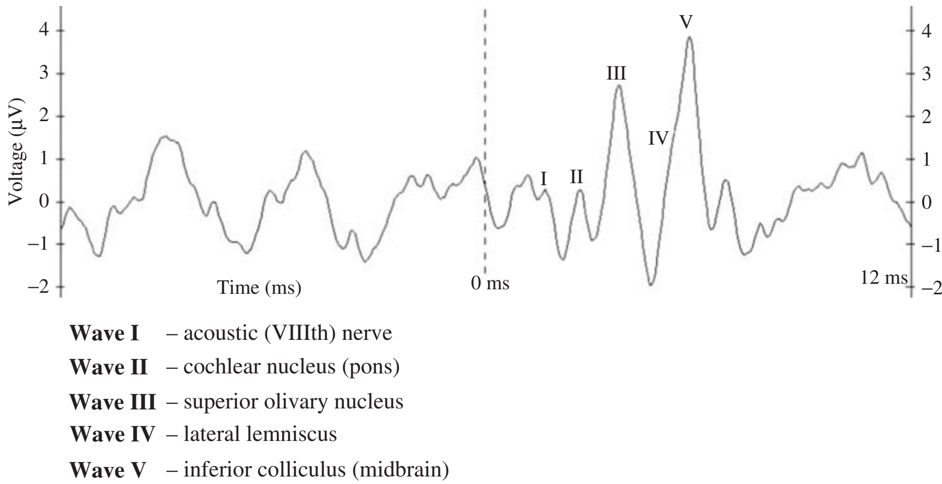


Figure 2.10 EEG signal showing an ABR Jewett waveform

Wavelet decomposition is a form of subband filtering and has many uses in DSP. By breaking the signal down into the frequency bands such as those illustrated in Figure 2.9, denoising can be performed by eliminating the coefficients representing the highest frequency components (e.g., levels D1 to D3) and then reconstructing the signal using the remaining coefficients. Naturally, this could also be used for data compression in a similar method to the DCT and has been applied to image compression.

Wavelet decomposition is also a powerful transform to use in analysis of medical signals. One example is the determination of hearing acuity using ABR signals from the EEG recordings as introduced in Section 2.3.1. The ABR response itself is a deterministic signal giving the same, time-locked response at the same stimulus for the same subject, and is often referred to as the Jewett waveform (Jewett 1970) given in Figure 2.10. The amplitude of the ABR is typically less than 1 μV and is hidden behind the background brain activity which may be as much as 100 μV. To extract the ABR, averaging of several thousand responses is often performed. As the background noise is white Gaussian in nature, averaging over a substantial number acts to drive the noise to zero. Meanwhile, the deterministic ABR maintains its presence, though some alteration to the morphology can be expected. Even with this noise reduction, it can be difficult to ascertain if the subject has heard or not, particularly in threshold cases. The response is time locked within 10 ms post-stimulus. Furthermore, it has key peaks in certain frequency bands, namely, 200, 500 and 900 Hz. Wavelet decomposition enables these key frequency bands to be extracted whilst maintaining the temporal information useful in isolating the regions where the key peaks of the response are expected. Figure 2.11 illustrates the wavelet coefficients from the D4 band (see Figure 2.9) for the section of the waveform before the stimulus and compare it to the post-stimulus section. This figure highlights how wavelet decomposition can be used to focus in on both the frequency and temporal regions of interest in a signal.

2.5 Filter Structures

2.5.1 Finite Impulse Response Filter

A simple finite impulse response (FIR) filter is given in Equation (2.11) where a_i are the coefficients needed to generate the necessary filtering response such as low-pass or high-pass and N is the

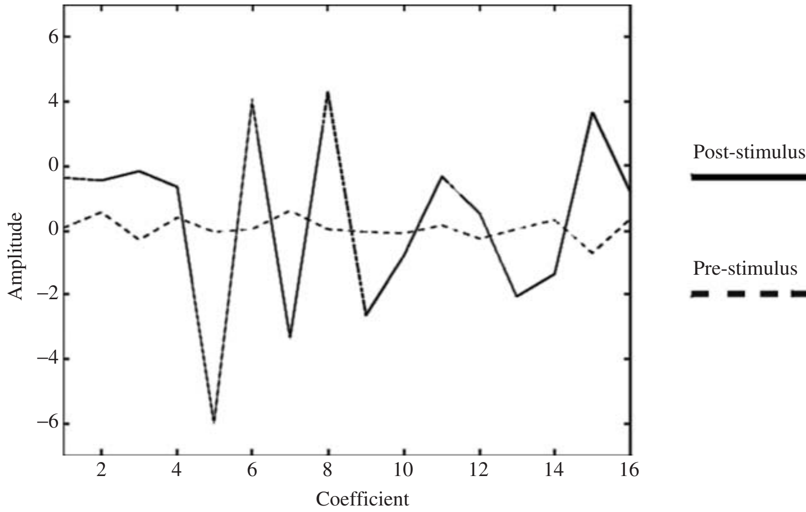


Figure 2.11 Wavelet coefficients for the D4 level

number of filter taps contained in the function. Typically, a_i and the input $x(n)$, will have finite length N, i.e. they will be non-zero for samples, $x(n), n = 0, 1, \ldots, N-1$.

The function given in Equation (2.12) can be represented using the classical signal flow graph (SFG) representation of Figure 2.12 for $N = 3$. In the classic form, the delay boxes of z^{-1} indicate a digital delay, the branches send the data to several output paths, labelled branches represent a multiplication by the variable shown and the black dots indicate summation functions. The block diagram form given in Figure 2.13 is also used where multiplication and additions is expressed by the function blocks shown.

$$y(n) = \sum_{i=0}^{N-1} a_i x(n-i) \tag{2.11}$$

$$y(n) = a_0 x(n) + a_1 x(n-1) + a_2 x(n-2) \tag{2.12}$$

A FIR filter exhibits some important properties including the following:

Superposition. Superposition holds if a filter produces an output $y(n) + v(n)$ from an input $x(n) + u(n)$ where $y(n)$ is the output produced from input $x(n)$ and $v(n)$ is the output produced from input $u(n)$.

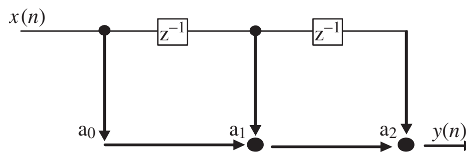


Figure 2.12 Original FIR filter SFG

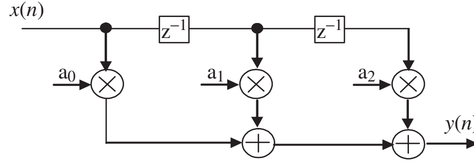


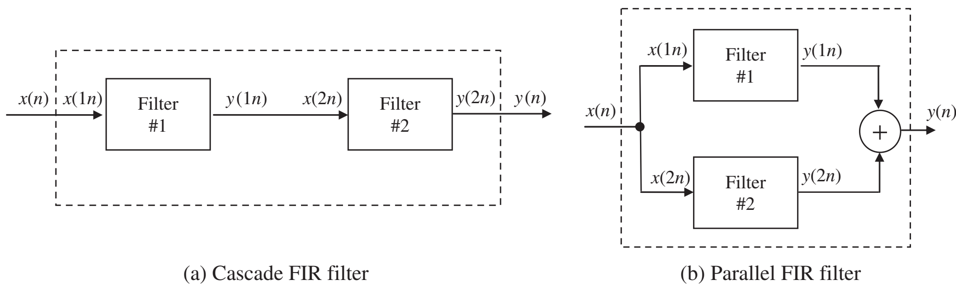
Figure 2.13 FIR filter SFG

Homogeneity. If a filter produces an output *ay(n)* from input *ax(n)*, then the filter is said to be homogeneous.

Shift invariance. A filter is shift invariant, if and only if, the input of $x(n+k)$ generates an output $y(n+k)$ where $y(n)$ is the output produced by $x(n)$.

If a filter exhibits all these properties then it is said to be a *linear*, *time-invariant* (LTI) filter. This property allows these filters to be cascaded, as shown in Figure 2.14(a) or connected in a parallel configuration as shown in Figure 2.14(b).

One basic way of developing a digital filter is to start with the desired frequency response, inverse filter it to get the impulse response, truncate the impulse response and then window the function to remove artifacts (Bourke 1999, Williams 1986). If we start with a typical low-pass function, as indicated in Figure 2.15, then translating this back to the time domain gives a typical *sinc* function with a starting point at zero. Realistically, we have to approximate this infinitely long filter with a finite number of coefficients, and given that it needs data from the future, time shift it so that it does not have negative values. If we can then successfully design the filter and transform



(a) Cascade FIR filter (b) Parallel FIR filter

Figure 2.14 FIR filter configurations

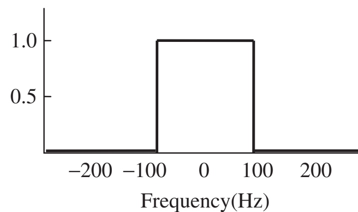


Figure 2.15 Low-pass filter response

it back to the frequency domain we get a ringing or rippling effect in the cut-off and passband frequency known as *rippling* and only a very gradual transition between passband and stopband regions, termed the *transition region*. The ripple is often called Gibbs phenomenum after Willard Gibbs who identified this effect in 1899.

We can reduce these effects by increasing the number of coefficients or taps to the filter, thereby allowing a better approximation of the filter, but at increased computation cost and also by using *windowing*. Indeed, it could be viewed that we were windowing the original frequency plot with a rectangular window, but there are other types most notably von Hann, Hamming and Kaiser windows (Lynn and Fuerst 1994, Williams 1986); these can be used to minimize rippling and transition in different ways and to different levels. The result of the design process is the determination of the filter length and coefficient values which best meet the requirements of the filter response. FIR filter implementations are relatively simple to understand as there is a straightforward relationship between the time and frequency domain. They have a number of additional advantages, including:

- linear phase, meaning that they delay the input signal, but do not distort its phase
- inherent stability
- can be implemented using finite arithmetic
- low sensitivity to quantization errors

2.5.2 Correlation

A related function employed in digital communications is called *correlation* and is given by the following computation:

$$y(n) = \sum_{i=-\infty}^{\infty} a_i x(n+i) \tag{2.13}$$

Given that the correlation is performed on a finite sequence, x_n, $n = 0, 1, \ldots, N-1$, this expression is given by

$$y(n) = \sum_{i=0}^{N-1} a_i x(n+i) \tag{2.14}$$

which is similar to the FIR filter expression of Equation (2.11) except that the order of the x_n data stream is reversed. The structures created for correlation are similar to those for FIR filters.

2.5.3 Infinite Impulse Response Filter

The main disadvantage of FIR filters is the high number of taps needed to realize some aspects of the frequency response, namely sharp cut-off, resulting in a high computational cost to achieve this performance. This can be overcome by using infinite impulse response (IIR) filters which use previous values of the output as indicated in Equation (2.15). This is best expressed in the transfer function expression given in Equation (2.16) and is shown in Figure 2.16.

$$y(n) = \sum_{i=0}^{N-1} a_i x(n-i) + \sum_{j=1}^{M-1} b_j y(n-j) \tag{2.15}$$

The design process is different from FIR filters and is usually achieved by exploiting the huge body of analogue filter designs; by transforming the s-plane representation of the analogue filter into the z-domain (Grant *et al.* 1989), a realistic digital filter implementation is achieved. A number

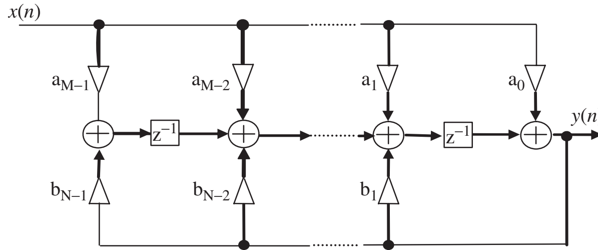


Figure 2.16 Direct form IIR filter

of design techniques such as the Impulse Invariant method (Williams 1986), the match z-transform and the Bilinear transform (Grant *et al.* 1989, Williams 1986). The resulting design gives a transfer function expression comprised of poles and zeroes as outlined by Equation (2.17). The main concern is to maintain stability which is achieved by ensuring that the poles are located within the unit circle. The location of these zeroes and poles have a direct relationship on the filter properties. For example, a pole on the unit circle with no zero to annihilate it will produce an infinite gain at a certain frequency (Meyer-Baese 2001).

$$H(z) = \frac{\sum_{i=0}^{N-1} a_i x_{n-i}}{1 - \sum_{j=1}^{M-1} b_j y_{n-j}} \tag{2.16}$$

$$H(z) = G\frac{(z-\xi_1)(z-\xi_2)\ldots(z-\xi_M)}{(z-\rho_1)(z-\rho_2)\ldots(z-\rho_N)} \tag{2.17}$$

Due to the feedback loops as shown in Figure 2.16, the structures are very sensitive to quantization errors, a feature which increases as the filter order grows. For this reason, filters are built from a concatenation of second-order IIR filters defined by Equation (2.18) leading to the structure of Figure 2.17 where each of block $H_z(j)$, $j = 1, 2, \ldots, N/2$ is a second-order IIR filter.

$$y(n) = a_0 x(n) + a_1 x(n-1) + a_2 x(n-2) + b_1 y(n-1) + b_2 y(n-2) \tag{2.18}$$

2.5.4 *Wave Digital Filters*

In addition to nonrecursive (FIR) and recursive (IIR) filters, a class of filter structures called wave digital filters (WDFs) are also of considerable interest as they possess a low sensitivity to coefficient variations. This is important as in IIR filter, this determines the level of accuracy to

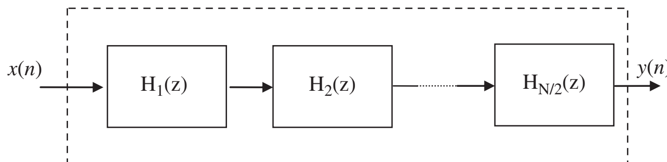


Figure 2.17 Cascade of second-order IIR filter blocks

which the filter coefficients have to be realized and has a direct correspondence to the dynamic range needed in the filter structure; adjusting internal wordlength sizes and filter performance is important from a hardware perspective as this will invariably affect throughput rate. This is largely because WDFs possess a low sensitivity to attenuation due to their inherent structure, thereby reducing the loss response due to changes in coefficient representation. This is important for many DSP applications for a number of reasons: it allows short coefficient representations to be used which meet the filter specification and which involve only a small hardware cost and; structures with low coefficient sensitivities also generate small round-off errors, i.e. errors that result as an effect of limited arithmetic precision within the structure (the issue of truncation and wordlength errors are discussed later). As with IIR filters, the starting principle is to generate low-sensitivity digital filters by capturing the low-sensitivity properties of the analogue filter structures.

WDFs represent a class of filters that are modelled on classical analogue filter networks (Fettweis and Nossek 1982, Fettweis *et al*. 1986, Wanhammar 1999) which are typically networks configured in the lattice or ladder structure. For circuits that operate on low frequencies where the circuit dimensions are small relative to the wavelength, the designer can treat the circuit as an interconnection of lumped passive or active components with unique voltages and currents defined at any point in the circuit, on the basis that the phase change between aspects of the circuit will be negligible. This allows a number of the circuit level design optimization techniques, such as Kirchoff's law, to be applied. However, at higher-frequency circuits, these assumptions no longer apply and the user is faced with solving Maxwell's equations. To avoid this, the designers can exploit the fact that the designer is solving the problems only at restricted places, such as the voltage and current levels at the terminals (Pozar 2005). By exploiting specific types of circuits such as transmission lines which have common electrical propagation times, circuits can then be treated as transmission line components treated as a distributed component characterized by its length, propagation constant and characteristic impedance.

The process of producing a WDF has been comprehensively covered by Fettweis *et al*. (1986). The main design technique is to generate filters using transmission line filters and relate these to classical filter structures with lumped circuit elements, thereby exploiting the well-known properties of these structures thereby termed a *reference filter*. The correspondence between the WDF and its reference filter is achieved by mapping the reference structure using a complex frequency variable, ψ termed the *Richards variable*, allowing the reference structure to be mapped effectively into the ψ domain. The use of reference structures allows all the inherent passivity and lossless features to be transferred into the digital domain, thereby achieving good filter performance and reducing the coefficient sensitivity to allow use of lower wordlengths. Work in Fettweis *et al*. (1986) give the simplest and most appropriate choice of ψ as the bilinear transform of the z-variable, given in Equation (2.19),

$$\psi = \frac{z-1}{z+1} = \tanh(\rho T/2) \qquad (2.19)$$

where ρ is the actual complex frequency. This variable has the property that the real frequencies ω correspond to real frequencies ϕ, accordingly to Equation (2.20):

$$\phi = \tan(\omega T/2), \rho = j\alpha, \psi = j\phi \qquad (2.20)$$

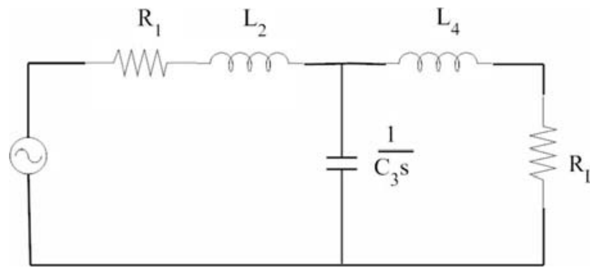
This implies that the real frequencies in the reference domain correspond to real frequencies in the digital domain. Other properties (Fettweis *et al*. 1986) ensure that the filter is causal. The basic principle used for WDF filter design is illustrated in Figure 2.18, taken from (Wanhammar 1999). The lumped element filter is shown in Figure 2.18(a) where the various passive components L_2s, $1/C_3s$ and L_4s, map to $R_2\psi$, R_3/ψ and $R_4\psi$, respectively in the analogous filter given in Figure 2.18(b). Equation (2.19) is then used to map the equivalent transmission line circuit to give the ψ domain filter in Figure 2.18(c).

WDF Building Blocks

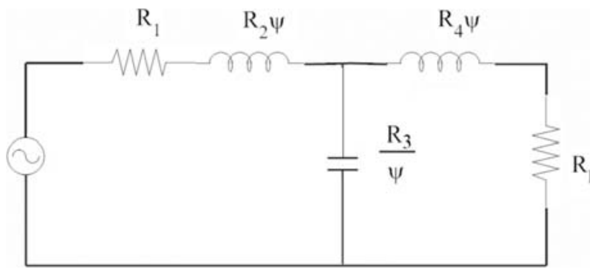
As indicated in Figure 2.18(c), the basic WDF configuration is based upon the various one-, two- and multi-port elements. Figure 2.19 gives the basic description of the two-port element. The network can be described by incident A_1, and reflected B_2 waves which are related to the port currents, I_1 and I_2, port voltages, V_1 and V_2 and port resistances, R_1 and R_2 as given below (Fettweis *et al*. 1986):

$$A_1 \cong V_1 + R_1 I_1 \tag{2.21}$$

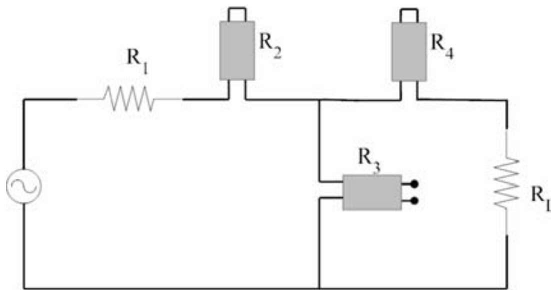
$$B_2 \cong V_2 + R_2 I_2 \tag{2.22}$$



(a) Reference lumped element filter



(b) ψ domain filter structure



(c) Resulting two port filter

Figure 2.18 Wave digital filter configuration (Wanhammar, 1999)

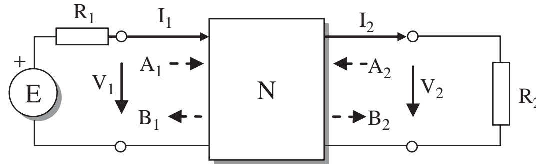


Figure 2.19 Basic description of a two-port adaptor

and the transfer function, S_{21} is given by Equation (2.23),

$$S_{21} = KB_2/A_1 \tag{2.23}$$

where K is given as:

$$K = \sqrt{R_1/R_2} \tag{2.24}$$

In their seminal paper, Fettweis *et al.* (1986) show that the loss α can be related to the circuit parameters, namely the inductances or capacitances, and frequency ω such that the loss is $\omega = \omega_0$, indicating that for a well-designed filter, the sensitivity of the attenuation is small through its passband, thus giving the earlier stated advantages of lower coefficient wordlengths.

The basic building blocks for the reference filters are a number of these common two-port and three-port elements or adaptors, as can be seen from the simple structure in Figure 2.18(c). A more fuller definition of these blocks is given in (Fettweis *et al.* 1986, Wanhammar 1999), but mostly they comprise multipliers and adders.

2.6 Adaptive Filtering

The material in Sections 2.2–2.5 has described both transforms and filtering algorithms that are used in a wide range of applications. Typically, these will be applied to provide transformations either to or from the frequency domain using the FFT or DCT, or to identify some features of a signal as in the EEG filtering example using filtering algorithms. However, there are a number of applications where the area of interest will not be known and a different class of filtering algorithms are required. These are known as adaptive algorithms; they are intriguing as they provide challenges from a high-speed implementation point-of-view. Indeed, a chapter is dedicated later in the book to the hardware implementation of an adaptive filter.

2.7 Basics of Adaptive Filtering

The basic function of a filter is to remove unwanted signals from those of interest. Obtaining the best design usually requires *a priori* knowledge of certain statistical parameters (such as the mean and correlation functions) within the useful signal. With this information, an optimal filter can be designed which minimizes the unwanted signals according to some statistical criterion. One popular measure involves the minimization of the mean square of the error signal, where the error is the difference between the desired response and the actual response of the filter. This minimization leads to a cost function with a uniquely defined optimum design for stationary inputs, known as a Wiener filter (Widrow and Hoff 1960). However, it is only optimum when the statistical characteristics of the input data match the *a priori* information from which the filter is designed, and is therefore inadequate when the statistics of the incoming signals are unknown or changing, i.e.

in a nonstationary environment. For this situation, a time-varying filter is needed to allow for these changes. An appropriate solution is an adaptive filter, which is inherently *self-designing* through the use of a *recursive* algorithm to calculate updates for the filter parameters. These updates are used to compute the taps of the new filter, the output of which is used with new input data to form the updates for the next set of parameters. When the input signals are stationary (Haykin 2001), the algorithm will converge to the optimum solution after a number of iterations, according to the set criterion. If the signals are non-stationary then the algorithm will attempt to track the statistical changes in the input signals, the success of which depends on its inherent convergence rate versus the speed at which statistics of the input signals are changing. Figure 2.20 gives the generalized structure of an adaptive filter. Here we see an input signal $x(n)$ fed into both the FIR filter and also the adaptive algorithm. The output from the adaptive FIR filter is $y_{\text{est}}(n)$, an estimation of the desired signal, $y(n)$. The difference between $y(n)$ and $y_{\text{est}}(n)$ gives an error signal $e(n)$. The adaptive algorithm uses this signal and the input signal $x(n)$ to calculate updates for the filter weights $w(n)$. This generalized adaptive filter structure can be applied to a range of applications, some of which will be discussed in the next section.

2.7.1 Applications of Adaptive Filters

Adaptive filters, because of their ability to operate satisfactorily in non-stationary environments, have become an important part of DSP applications where the statistics of the incoming signals are unknown or changing. Some examples are in channel equalization, echo cancellation or adaptive beamforming. The basic function comes down to the adaptive filter performing a range of different tasks, namely, system identification, inverse system identification, noise cancellation and prediction. For each of these applications, different formats are used of the general filter structure shown in Figure 2.20. This section will give detail of each of these adaptive filtering set-ups with example applications.

Adaptive filtering applied to system identification is illustrated in Figure 2.21. Here, the adaptive filter aims to model the unknown system H, where H has an impulse response given by $h(n)$, for $n = 0, 1, 2, 3, \ldots, \infty$, and zero for $n < 0$. The $x(n)$ input to H is also fed into the adaptive filter. As before, an error signal $e(n)$ is calculated and this signal is used to calculate an update for the filter weights, $w(n)$, thus calculating a closer estimation to the unknown system, $h(n)$. In telecommunications, an echo can be present on the line due to a impedance mismatch in hybrid components on the public switched telephone networks. In this case, the adaptive system is trained to model the unknown echo path so that in operation, the filter can negate effects of this path, by synthesizing the resounding signal and then subtracting it from the original received signal.

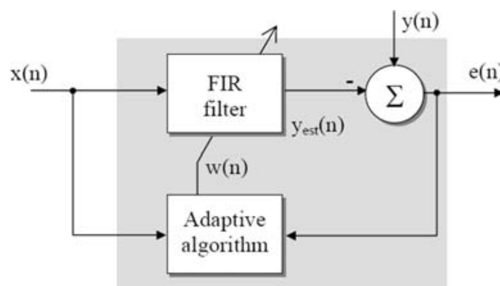


Figure 2.20 General adaptive filter system

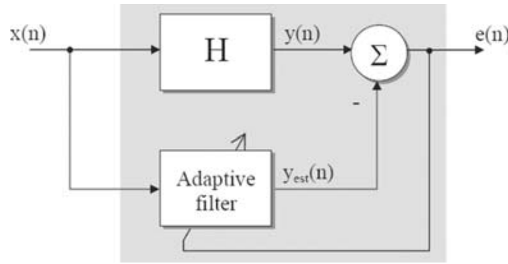


Figure 2.21 System indentification

Figure 2.22 shows the inverse system identification. Similarly to the example in Figure 2.21, the adaptive filter is trying to define a model, however, in this case it is trying to model the inverse of an unknown system H, thereby negating the effects of this unknown system. One such application is channel equalization (Drewes *et al*. 1998) where the inter-symbol interference and noise within a transmission channel are removed by modelling the inverse characteristics of the contamination within the channel.

Figure 2.23 shows the adaptive filter used for prediction. A perfect example of this application is in audio compression for speech for telephony. The adaptive differential pulse code modulation (ADPCM) algorithm tries to predict the next audio sample. Then only the difference between the prediction and actual value is coded and transmitted. By doing this, the data rate for speech can be halved to 32 kbps while maintaining 'toll quality'. The international standard (International Telecommunications Union) for ADPCM defines the structure as an IIR two-pole, six-zero adaptive predictor (ITU-T 1990).

The structure of adaptive noise cancellation is given in Figure 2.24. Here the structure is slightly different. The reference signal in this case is the data $s(n)$ corrupted with a noise signal $\nu(n)$. The input to the adaptive filter is a noise signal $\nu'(n)$ that is strongly correlated with the noise $\nu(n)$, but uncorrelated with the desired signal $s(n)$. There are many applications in which this may be used. It can be applied to echo cancellation for both echoes caused by hybrids in the telephone networks,

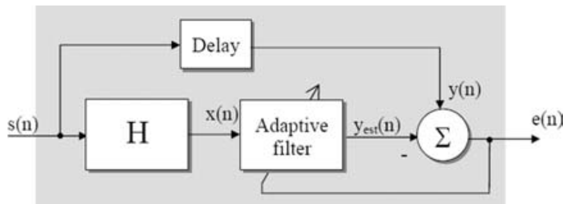


Figure 2.22 Inverse system indentification

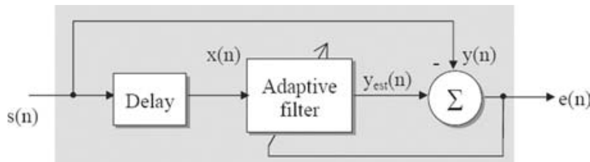


Figure 2.23 Prediction

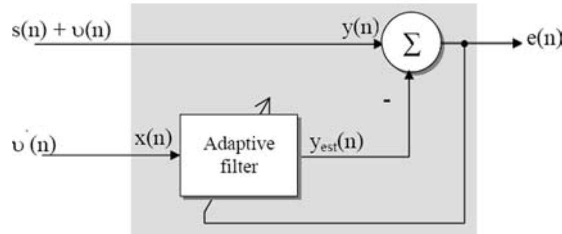


Figure 2.24 Noise cancellation

but also for acoustic echo cancellation in hands-free telephony. Within medical applications, noise cancellation can be used to remove contaminating signals from electrocardiograms (ECG). A particular example is the removal of the mother's heartbeat from the ECG trace of an unborn child (Baxter *et al*. 2006).

Adaptive beamforming is another key application and can be used for noise cancellation (Litva and Lo 1996, Moonen and Proudler 1998, Ward *et al*. 1986). The function of a typical adaptive beamformer is to suppress signals from every direction other than the desired 'look direction' by introducing deep nulls in the beam pattern in the direction of the interference. The beamformer output is a weighted combination of signals received by a set of spatially separated antennae, one primary antenna and a number of auxiliary antennae. The primary signal constitutes the input from the main antennae, which has high directivity. The auxiliary signals contain samples of interference threatening to swamp the desired signal. The filter eliminates this interference by removing any signals in common with the primary input signal, i.e. those that correlate strongly with the reference noise. The input data from the auxiliary and primary antennae is fed into the adaptive filter, from which the weights are calculated. These weights are then applied on the delayed input data to produce the output beam. This is considered in detail in Chapter 12.

2.7.2 Adaptive Algorithms

The computational ingenuity within adaptive filtering are the algorithms used to calculate the updated filter weights. Two conflicting algorithms dominate the area, the *recursive least-squares* (RLS) algorithm and the *least-mean-squares* (LMS) algorithm. The RLS algorithm is a powerful technique derived from the method of least-squares. It offers greater convergence rates than its rival LMS algorithm, but this gain is at a cost of increased computational complexity, a factor that has hindered its use in real-time applications. The LMS algorithm offers a very simplistic yet powerful approach, giving good performance under the right conditions (Haykin 2001). However, its limitations lie with its sensitivity to the condition number of the input data matrix as well as slow convergence rates.

Filter coefficients may be in the form of tap weights, reflection coefficients or rotation parameters depending on the filter structure, i.e. transversal, lattice or systolic array respectively, (Haykin 1986). However, the LMS and RLS algorithms can be applied to the basic structure of a transversal filter given in Figure 2.13, consisting of a linear combiner which forms a weighted sum of the system inputs, $x(n)$, and then subtracts them from the desired signal, $y(n)$, to produce an error signal, $e(n)$, as given in Equation (2.25). In Figure 2.20, $w(n)$ are the adaptive weight vectors.

$$e(n) = y(n) - \sum_{i=0}^{N-1} W_i x_{n-i} \tag{2.25}$$

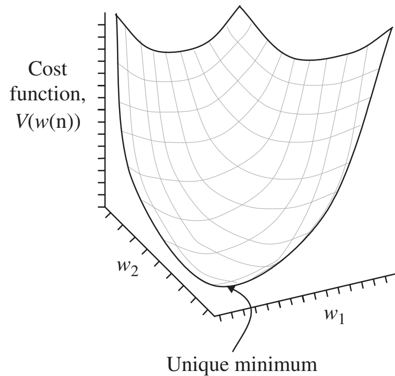


Figure 2.25 Error surface of a two-tap transversal filter (Haykin 2001)

There is no distinct technique to determine the optimum adaptive algorithm for a specific application. The choice comes down to a balance in the range of characteristics defining the algorithms, such as:

- rate of convergence, i.e. the rate at which the adaptive algorithm reaches within a tolerance of an optimum solution
- steady-state error, i.e. the proximity to an optimum solution
- ability to track statistical variations in the input data
- computational complexity
- ability to operate with ill-conditioned input data
- sensitivity to variations in the wordlengths used in the implementation

2.7.3 LMS Algorithm

The LMS algorithm is a stochastic gradient algorithm, which uses a fixed step-size parameter to control the updates to the tap weights of a transversal filter, (Widrow and Hoff 1960) as shown in Figure 2.20. The algorithm aims to minimize the mean-square error, the error being the difference in $y(n)$ and $y_{est}(n)$. The dependence of the mean-square error on the unknown tap weights may be viewed as a multidimensional paraboloid referred to as the error surface (depicted in Figure 2.25 for a two-tap example) (Haykin 2001). The surface has a uniquely defined minimum point defining the tap weights for the optimum Wiener solution, (defined by the Wiener–Hopf equations detailed in the next subsection). However, in the non-stationary environment this error surface is continuously changing, thus the LMS algorithm needs to be able to track the bottom of the surface. The LMS algorithm aims to minimize a cost function, $V(w(n))$, at each time step n, by a suitable choice of the weight vector $w(n)$. The strategy is to update the parameter estimate proportional to the instantaneous gradient value, $\mathrm{d}V(w(n))/\mathrm{d}w(n)$, so that:

$$w(n+1) = w(n) - \mu \frac{\mathrm{d}V(w(n))}{\mathrm{d}w(n)} \tag{2.26}$$

where μ is a small positive step size and the minus sign ensures that the parameter estimates descend the error surface.

The cost function, $V(w(n))$, which minimizes the mean-square error, results in the following recursive parameter update equation:

$$w(n+1) = w(n) - \mu x(n)(y(n) - y_{\mathrm{est}}(n)) \tag{2.27}$$

The recursive relation for updating the tap weight vector Equation (2.26) may be rewritten as:

$$w(n+1) = w(n) - \mu x(n)(y(n) - x^{\mathrm{T}}(n)w(n)) \tag{2.28}$$

which can be represented as filter output (Equation 2.29), estimation error (Equation 2.30) and tap weight adaptation (Equation 2.31).

$$y_{\mathrm{est}}(n) = w^{\mathrm{T}}(n)x(n) \tag{2.29}$$

$$e(n) = y(n) - y_{\mathrm{est}}(n) \tag{2.30}$$

$$w(n+1) = w(n) + \mu x(n)e(n) \tag{2.31}$$

The LMS algorithm requires only $2N + 1$ multiplications and $2N$ additions per iteration for an N tap weight vector. Therefore it has a relatively simple structure and the hardware is directly proportional to the number of weights.

2.7.4 RLS Algorithm

In contrast RLS is a computationally complex algorithm derived from the method of least-squares (LS) in which the cost function, $J(n)$, aims to minimize the sum of squared errors, as given in Equation (2.32):

$$J(n) = \sum_{i=0}^{N-1} |e(n-i)|^2 \tag{2.32}$$

Substituting (Equation 2.25) into (Equation 2.32) gives:

$$J(n) = \sum_{i=0}^{N-1} \left| y(n) - \sum_{i=0}^{N-1} w_k x(n-i) \right|^2 \tag{2.33}$$

Converting from the discrete time domain to a matrix–vector form simplifies the representation of the equations. This is achieved by considering the data values from N samples, so that Equation (2.25) becomes:

$$e(n) = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} - \begin{bmatrix} \underline{x}_1^{\mathrm{T}} \\ \underline{x}_2^{\mathrm{T}} \\ \vdots \\ \underline{x}_N^{\mathrm{T}} \end{bmatrix} \begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_N \end{bmatrix} \tag{2.34}$$

which may be expressed as:

$$e(n) = y(n) - X(n)w(n) \tag{2.35}$$

The cost function $J(n)$ may then be represented in matrix form as:

$$J(n) = e(n)^{\mathrm{T}}e(n) = (y(n) - X(n)w(n))^{\mathrm{T}}(y(n) - X(n)w(n)) \tag{2.36}$$

This is then multiplied out and simplified to give:

$$J(n) = y^{\mathrm{T}}(n) - 2y^{\mathrm{T}}(n)X(n)w(n) + w^{\mathrm{T}}(n)X^{\mathrm{T}}(n)X(n)w(n) \tag{2.37}$$

where $X^{\mathrm{T}}(n)$ is the transpose of $X(n)$ and $y^{\mathrm{T}}(n)$ is the transpose of $y(n)$. To find the optimal weight vector, this expression is differentiated with respect to $w(n)$ and solved to find the weight vector that will drive the derivative to zero. This results in a least-squares weight vector estimation, $w_{LS}(n)$, which is derived from the above expression and can be expressed in matrix format as:

$$w_{LS}(n) = (X^{\mathrm{T}}(n)X(n))^{-1}X^{\mathrm{T}}y(n) \tag{2.38}$$

These are referred to as the Wiener–Hopf normal equations (Equations 2.39–2.41)

$$w_{LS}(n) = \phi(n)^{-1}\theta(n) \tag{2.39}$$

$$\phi(n) = X^{\mathrm{T}}(n)X(n) \tag{2.40}$$

$$\theta(n) = X^{\mathrm{T}}(n)y(n) \tag{2.41}$$

where $\phi(n)$ is the correlation matrix of the input data $X(n)$ and $\theta(n)$ is the cross-correlation vector of the input data $X(n)$ with the desired signal vector $y(n)$. By assuming that the number of observations is larger than the number of weights, a solution can be found since there are more equations than unknowns.

The LS solution given so far is performed on blocks of sampled data inputs. This solution can be implemented recursively, using the RLS algorithm, where the LS weights are updated with each new set of sample inputs. Continuing this adaptation through time would effectively perform the LS algorithm on an infinitely large window of data and would therefore only be suitable for a stationary system. A weighting factor may be included within the LS solution for application in nonstationary environments. This factor assigns greater importance to the more recent input data, effectively creating a moving window of data on which the LS solution is calculated. The forgetting factor β is included in the LS cost function (from Equation 2.36) as:

$$J(n) = \sum_{i=0}^{N-1} \beta(n-i)e^2(i) \tag{2.42}$$

where $\beta(n-i)$ is defined as $0 < \beta(n-i) \leq 1$, $i = 1, 2 \ldots, N$. One form of the forgetting factor is the exponential forgetting factor:

$$\beta(n-i) = \lambda^{n-i} \tag{2.43}$$

where $i = 1, 2, \ldots, N$, and λ is a positive constant with a value close to, but less than 1. Its value is of particular importance as it determines the length of data window that is used and will effect the performance of the adaptive filter. The inverse of $(1 - \lambda)$ gives a measure of the *memory* of the algorithm. The general rule is that the longer the memory of the system, the faster the convergence and the smaller the steady-state error. However, the window length is limited by the rate of change

in the statistics of the system. Applying the forgetting factor to the Wiener–Hopf normal equations (Equations 2.39–2.41), the correlation matrix and the cross-correlation matrix become:

$$\phi(n) = \sum_{i=0}^{n} \lambda^{n-1} \underline{x}(i)\underline{x}^{\mathrm{T}}(i) \tag{2.44}$$

$$\theta(n) = \sum_{i=0}^{n} \lambda^{n-1} \underline{x}(i)\underline{y}(i) \tag{2.45}$$

The recursive representations are then expressed as:

$$\phi(n) = \left[\sum_{i=1}^{n-1} \lambda^{n-i-1} \underline{x}(i)\underline{x}^{T}(i)\right] + \underline{x}(n)\underline{x}^{\mathrm{T}}(n) \tag{2.46}$$

or more concisely as:

$$\phi(n) = \lambda\phi(n-1) + \underline{x}(n)\underline{x}^{\mathrm{T}}(n) \tag{2.47}$$

Likewise, $\theta(n)$ can be expressed as:

$$\theta(n) = \lambda\theta(n-1) + \underline{x}(n)\underline{y}(n) \tag{2.48}$$

Solving the Wiener–Hopf normal equations to find the LS weight vector requires the evaluation of the inverse of the correlation matrix, as highlighted by the example matrix vector expression below (Equation 2.49):

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \underbrace{\left[\begin{bmatrix} X_{11}X_{12}X_{13} \\ X_{21}X_{22}X_{23} \\ X_{31}X_{32}X_{33} \end{bmatrix}^{T} \begin{bmatrix} X_{11}X_{12}X_{13} \\ X_{21}X_{22}X_{23} \\ X_{31}X_{32}X_{33} \end{bmatrix}\right]^{-1}}_{\text{correlation matrix}} \bullet \underbrace{\left[\begin{bmatrix} X_{11}X_{12}X_{13} \\ X_{21}X_{22}X_{23} \\ X_{31}X_{32}X_{33} \end{bmatrix}^{T} \begin{bmatrix} y_{11} \\ y_{12} \\ y_{13} \end{bmatrix}\right]}_{\text{cross-correlation matrix}} \tag{2.49}$$

The presence of this matrix inversion creates an implementation hindrance in terms of both numerical stability and computational complexity. For instance the algorithm would be subject to numerical problems if the correlation matrix became singular. Also, calculating the inverse for each iteration requires an order of complexity N^3, compared with a complexity of order N for the LMS algorithm. There are two particular methods to solve the LS solution recursively without the direct matrix inversion which reduce this complexity to order N^2. The first technique, referred to as the standard RLS algorithm, recursively updates the weights using the matrix inversion lemma. The alternative and very popular solution performs a set of orthogonal rotations, e.g. Givens rotations (Givens 1958), on the incoming data, transforming the square data matrix into an equivalent upper triangular matrix (Gentleman and Kung 1981). The weights can then be calculated by back-substitution. This method, known as QR decomposition (performed using one of a range of orthogonal rotation methods such as Householder transformations or Givens rotations), has been the basis for a family of numerically stable and robust RLS algorithms (Cioffi 1990, Cioffi and Kailath 1984, Dohler 1991, Hsieh *et al*. 1993, Liu *et al*. 1990, 1992, McWhirter 1983, McWhirter *et al*. 1995, Rader and Steinhardt 1986, Walke 1997). There are versions of the RLS algorithm known as Fast RLS algorithms. These manipulate the redundancy within the system to reduce the complexity to the order of N.

2.8 Conclusions

The chapter has given a brief grounding in DSP terminology and covered some of the common DSP algorithms, ranging from transforms such as the DCT, FFT and DWT through to basic filter structures such as FIR and IIR filters right through to adaptive filters such as LMS and RLS filters. There are of course, a much wider range of algorithms, but the purpose of the chapter was to cover the salient points of these algorithms as many are used in practical design examples later in the book. A more detailed treatment has been given to the RLS filter structure as a chapter is dedicated to the creation of a complex core later in the book (Chapter 12).

References

Baxter P, Spence G and McWhirter J (2006) Blind signal separation on real data: tracking and implementing *Proc. ICA-2006*, pp. 327–334, Charleston, USA.

Bourke P (1999) Fourier method of designing digital filters. Web publication downloadable from http://local.wasp.uwa.edu.au/ pbourke/other/filter/.

Cioffi J (1990) The fast householder filters rls adaptive algorithm RLS adaptive filter. *IEEE Proc. Int. Conf. on Acoustics, Speech and Signal Processing*, pp. 1619–1621.

Cioffi JM and Kailath T (1984) Fast recursive-least-square, transversal filters for adaptive filtering. *IEEE Trans. Acoustics, Speech, Signal Processing* **ASSP-32**(2), 998–1005.

Daubechies I (1992) Ten lectures on wavelets. *CBMS-NSF Regional Conference Series in Applied Mathematics*.

Dohler R (1991) Squared Givens' rotations. *IMA J. of Numerical Analysis* **II**, 1–5.

Drewes C, Hasholzner R and Hammerschmidt JS (1998) On implementation of adaptive equalizers for wireless atm with an extended QR-decomposition-based RLS-algorithm. *Proc. Int. Conf. on Acoustics, Speech and Signal Processing*, pp. 3445–3448.

Fettweis A and Nossek J (1982) On adaptors for wave digital filter. *IEEE Trans. Circuits and Systems* **CAS-29**(12), 797–806.

Fettweis A, Gazsi L and Meerkotter K (1986) Wave digital filter: Theory and practice. *Proc. the IEEE* **74**(2), 270–329.

Gentleman WM and Kung HT (1981) Matrix triangularisation by systolic array. *Proc. SPIE (Real-Time Signal Processing IV)* **298**, 298–303.

Givens W (1958) Computation of plane unitary rotations transforming a general matrix to triangular form. *J. Soc. Ind. Appl. Math* **6**, 26–50.

Grant P, Cowan C, Mulgrew B and Dripps JH (1989) *Analogue and Digital Signal Processing and Coding*. Chartwell-Bratt Inc., Sweden.

Haykin S (2001) *Adaptive Filter Theory*. Prentice Hall, Englewood Cliffs, NJ.

Hsieh SF, Liu KJR and Yao K (1993) A unified approach for QRD-based recursive least-squares estimation without square roots. *IEEE Trans. Signal Processing* **41**(3), 1405–1409.

ITU-T (1990) Recommendation g.726,40, 32, 24, 16 kbit/s adaptive differential pulse code modulation (ADPCM).

Jewett D (1970) Volume conducted potentials in response to auditory stimuli as detected by averaging in the cat. *Electroencephalography and Clinical Neurophysiology (Suppl.)* **28**, 609–618.

Litva J and Lo TKY (1996) *Digital Beamforming in Wireless Communications*. Artec House, Norwood, MA.

Liu KJR, Hsieh S and Yao K (1990) Recursive LS filtering using block householder transformations. *Proc. Int. Conf. on Acoustics, Speech and Signal Processing*, pp. 1631–1634.

Liu KJR, Hsieh S and Yao K (1992) Systolic block householder transformations for rls algorithm with two-level pipelined implementation. *IEEE Trans. On Signal Processing* **40**(4), 946–958.

Lynn P and Fuerst W (1994) *Introductory Digital Signal Processing with Computer Applications*. John Wiley & Sons, Chicester.

Mallat SG (1989) A theory for multiresolution signal decomposition: the wavelet representation. *IEEE Trans. Pattern Anal. Machine Intelligence* **11**(7), 674–693.

McWhirter JG (1983) Recursive least squares minimisation using systolic array. *Proc. SPIE (Real-Time Signal Processing IV)* **431**, 105–112.

McWhirter JG, Walke RL and Kadlec J (1995) Normalised givens rotations for recursive least squares processing. *Proc. VLSI Signal Processing, VIII*, pp. 323–332.

Meyer-Baese U (2001) *Digital Signal Processing with Field Programmable Gate Arrays*. Springer, Germany.

Moonen M and Proudler IK (1998) MDVR beamforming with inverse updating. *Proc. EUSIPCO*, pp. 193–196.

Nyquist H (2002) Certain topics in telegraph transmission theory. *American Telephone and Telegraph Co. New York, NY* **90**(2), 280–305.

Omondi AR (1994) *Computer Arithmetic Systems*. Prentice Hall, New York.

Pozar DM (2005) *Microwave Engineering*. John Wiley & Sons, Inc., USA.

Rabiner L and Gold B (1975) *Theory and Application of Digital Signal Processing*. Prentice Hall, New York.

Rader CM and Steinhardt AO (1986) Hyperbolic householder transformations, definition and applications. *Proc. Int. Conf. on Acoustics, Speech and Signal Processing*, **11**, 2511–2514.

Shannon CE (1949) Communications in the presence of noise. *Proc. IRE*, **37**, 10–21.

Walke RL (1997) *High Sample Rate Givens Rotations for Recursive Least Squares*. PhD Thesis, University of Warwick.

Wanhammar1999 L (1999) *DSP Integrated Circuits*. Academic Press, San Diego.

Ward CR, Hargrave PJ and McWhirter JG (1986) A novel algorithm and architecture for adaptive digital beamforming. *IEEE Trans. On Antennas and Propagation* **AP-34**(3), 338–346.

Widrow B and Hoff CSJ (1960) Adaptive switching circuits. *IRE WESCON Conv. Rec.*, pp. 96–104.

Williams C (1986) *Designing digital filters*. Prentice Hall, New York.

3

Arithmetic Basics

3.1 Introduction

The choice of arithmetic has always been a key aspect for DSP implementation as it not only affects algorithmic performance, but also can impact system performance criteria, specifically area, speed and power consumption. For a DSP implementation on processor platforms, the choice of arithmetic becomes one of selecting the suitable platform, typically either a floating-point implementation or a fixed-point realization, with a subsequent choice of suitable wordlength for the fixed-point wordlength. However, with FPGA platforms, the choice of arithmetic can have a much wider impact on the performance cost right through the design process; though to be fair, architectural decisions made by FPGA vendors which can be seen in Chapter 5, tend to dominate arithmetic choice. Nonetheless, it is therefore worth considering and understanding arithmetic representations, in a little more detail.

A key requirement of DSP implementations is the availability of suitable processing elements, specifically adders and multipliers; however, some DSP algorithms, particularly adaptive filters, also require dedicated hardware for performing division and square root. The realization of these functions and indeed the choice of number systems, can have a major impact on hardware implementation quality. For example, it is well known that different DSP application domains, i.e. image processing, radar and speech, can have different levels of bit toggling not only in terms of the number of transitions, but also in the toggling of specific bits (Chandrakasan and Brodersen 1996). More specifically, the signed bit in speech input, can toggle quite often, as data oscillates around zero whereas in image processing, the input typically is all positive. In addition, different applications can have different toggling activity in their lower significant bits (Chandrakasan and Brodersen 1996). For this reason, it is important that some basics of computer arithmetic are covered, specifically number representation as well as the implementation choices for some common arithmetic functions, namely adders and multipliers. However, these are not covered in great detail as the reality is that in the case of addition and multiplication, dedicated hardware is becoming available on FPGA and thus for many applications, the lowest area, fastest speed and lowest power implementations will be based on *these* hardware elements.

Whilst adders and multipliers are vital for DSP systems, it is important to concentrate on division and square root operations as these are required in many, more complex DSP functions. These are covered in some detail here, and a brief comparison of the various methods used to implement the operations, included. Dynamic range is a key issue in DSP, therefore the data representations, namely fixed- and floating-point, are important. A basic description along with a review of the notation is included.

FPGA-based Implementation of Signal Processing Systems R. Woods, J. McAllister, G. Lightbody and Y. Yi
© 2008 John Wiley & Sons, Ltd

The chapter is organized as follows. In Section 3.2, some basics of computer arithmetic are covered, including the various forms of number representations with mention of more advanced representations such as signed digit number representations (SDNRs), logarithmic number systems (LNS), residue number systems (RNS) and the coordinate rotation digital computer (CORDIC). Fixed- and floating-point representations are covered in Section 3.3. Section 3.4 gives a brief introduction to the implementation of adders and multipliers with some discussion on the implementation of more complex arithmetic operations that are useful in DSP systems, namely division and square root. The chapter finishes with a discussion of some key issues including details of fixed- and floating-point implementations and a highlight on some of the other issues in terms of arithmetic and its representation.

3.2 Number Systems

From our early years, we have been taught to deal with decimal representations in terms of calculating values, but the evolution of transistor technology inferred the adoption of binary number systems as a more natural representation for DSP systems. Initially, the section starts with a basic treatment of conventional number systems, explaining signed magnitude and one's complement, but concentrating on two's complement as it is the most popular representation currently employed. Alternative number systems are briefly reviewed as indicated in the introduction, as they have been applied in some FPGA-based DSP systems.

3.2.1 Number Representations

If N is an $(n + 1)$-bit unsigned number, then the unsigned representation of Equation (3.1) applies:

$$N = \sum_{i=0}^{n} x_i 2^i \tag{3.1}$$

where x_i is the ith binary bit of N and x_0 and x_n are *least significant bit* (*lsb*) and *most significant bit* (*msb*) respectively.

Signed Magnitude

In signed magnitude systems, the $n - 1$ lower significant bits represent the magnitude, and the msb, x_n bit, represents the sign. This is best represented pictorially in Figure 3.1(a), which gives the number wheel representation for a 4-bit word. In the signed magnitude notation, the magnitude of the word is decided by the three lower significant bits, and the sign determined by the sign bit, i.e. *msb*. However, this representation presents a number of problems. First, there are two representations of 0 which must be resolved by any hardware system, particularly if 0 is used to trigger any event, e.g. checking equality of numbers. As equality is normally achieved by checking bit-by-bit, this complicates the hardware. Lastly, operations such as subtraction are more complex, as there is no way to check the sign of the resulting value, without checking the size of the numbers and organizing accordingly.

One's Complement

In one's complement systems, the inverse of the number is obtained by inverting, i.e. one's complementing the bits of the original word. The conversion is given in Equation (3.2) for an n-bit word, and the pictorial representation for a 4-bit binary word given in Figure 3.1(b). The problem

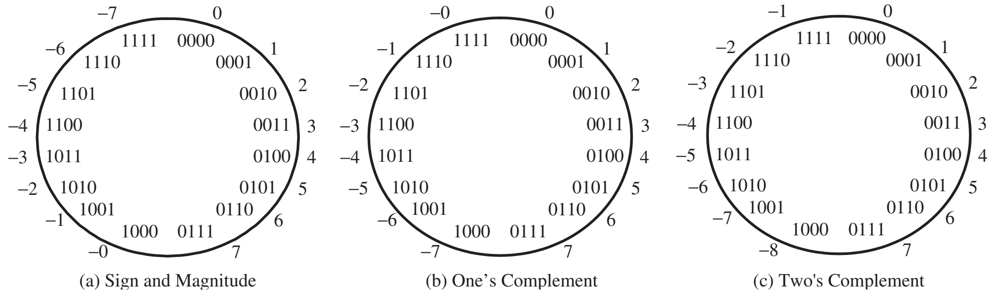


Figure 3.1 Number wheel representation of 4-bit number

still exists of two representations of 0 and a correction needs to be carried out when performing one's complement subtraction (Omondi 1994).

$$\overline{N} = (2^n - 1) - N \tag{3.2}$$

Two's Complement

In two's complement systems, the inverse of the number is obtained by inverting the bits of the original word and adding 1. The conversion is given in Equation (3.3) and the pictorial representation for a 4-bit binary word given in Figure 3.1(c). Whilst this may seem less intuitively obvious than the previous two approaches, it has a number of advantages: there is a single representation for 0, addition and more importantly subtraction, can be performed readily in hardware and, if the number stays within the range, overflow can be ignored in the computation. For these reasons, two's complement has become the dominant number system representation.

$$\overline{N} = 2^n - N \tag{3.3}$$

Signed Digit Number Representations

SDNRs were originally developed by Avizienis (1961), as a means to break carry propagation chains in arithmetic operations. A signed binary number representation (SBNR) was successfully applied by a number of authors (Andrews 1986, Knowles *et al.* 1989) in the high-speed design of circuits for arithmetic processing, digital filtering and Viterbi decoding functionality. By allowing a negative as well as a positive digit, means that a value can have a number of redundant representations. By exploiting this, Avizienis (Avizienis 1961) was able to demonstrate a system for performing parallel addition without the need for carry propagation. Of course, the redundant representation had to be converted back to binary, but several techniques were developed to achieve this (Sklansky 1960).

SBNR representation necessitated the development of a binary set for the SDNR digit, x where $x \in (-1,0,1)$ or strictly speaking $(\overline{1},0,1)$ where $\overline{1}$ represents -1. This is typically encoded by two bits, namely a *sign bit*, x_s and a *magnitude bit*, x_m bit as shown in Table 3.1. A more interesting assignment is the $(+,-)$ scheme where a SBNR digit is encoded as (x^+, x^-) where $x = x^+ + (x^- - 1)$. Alternatively this can be thought of as $x^- = 0$ implying -1 and $x^- = 1$ implying 0 and $x^+ = 0$ and $x^+ = 1$ implying 0 and 1, respectively. The key advantage of this approach is that it provides the ability to construct generalized SBNR adders from conventional adder blocks,

Table 3.1 SDNR encoding

SDNR digit	SDNR representations			
	Sig-and-mag		+/− coding	
x	x_s	x_m	x^+	x^-
0	0	0	0	1
1	0	1	1	0
$\overline{1}$	0	1	0	1
0 or X	1	0	1	0

and it is the key to the creation of high-speed multipliers. Whilst SDNR representations offer performance advantages from a speed perspective, conversion to and from, binary becomes an issue and dedicated circuitry is needed (Sklansky 1960); in addition, performing simple operations such as comparison becomes an issue due to the redundant representation of the internal data.

Other Representations

There are a number of other number representations, including logarithmic number representations (LNS) (Muller 2005), residue number representations (RNS) (Soderstrand *et al*. 1986) and the coordinate rotation digital computer (CORDIC) (Volder 1959, Walther 1971).

In LNS, a number x is represented as a fixed-point value i as given by Equation (3.4).

$$i = log_2 |x| \tag{3.4}$$

where extra bits are used to represent the sign of x and the special case of $x = 0$. A major advantage of the LNS is that multiplication and division in the linear domain is simply replaced by addition or subtraction in the log domain. However, the operations of addition and subtraction are more complex. In (Collange *et al*. 2006), the development of a LNS floating-point library is described and it is shown how it can be applied to some arithmetic functions and graphics applications.

RNS representations are useful in processing large integer values and therefore have application in computer arithmetic systems, and also in some DSP applications (see later), where there is a need to perform large integer computations. In RNS, an integer is converted into a number which is a N-tuple of smaller integers called *moduli*, given as $(m_N, m_{N-1}, \ldots, m_1)$. An integer X is represented in RNS by an N-tuple $(x_N, x_{N-1}, \ldots, x_1)$ with x_i is a non-negative integer, satisfying the following

$$X = m_i \cdot q_i + x_i \tag{3.5}$$

where q_i is the largest integer such that $0 \le x_i \le (m_i - 1)$ and the value x_i is known as the residue of the X modulo m_i. The main advantage of RNS is that additions, subtractions and multiplications are inherently carry-free due to the translation into the format. Unfortunately, other arithmetic operations such as division, comparison and sign detection are very slow and this has hindered the broader application of RNS. For this reason, the work has largely been applied to DSP operations that involve a lot of multiplications and additions such as FIR filtering and transforms such as the FFT and DCT (Soderstrand *et al*. 1986).

The CORDIC algorithm was originally proposed by Volder (1959). The algorithm makes it possible to perform rotations using only shift and add operations. This makes it attractive for computing trigonometric operation such as sine and cosine and also for multiplying or dividing numbers,

although Walther (1971) made it applicable to hyperbolic functions, logarithms, exponentials and square roots and for the first time, presented a unified algorithm for the three coordinate systems, namely linear, circular and hyperbolic. In CORDIC implementation, the reduced computational load in performing rotations (Takagi *et al*. 1991) means that it has been used for some DSP applications, particularly those implementing matrix triangularization (Ercegovac and Lang 1990) and RLS adaptive filtering (Ma *et al*. 1997) as this latter application requires rotation operations.

These represent dedicated implementations and the restricted application domain of the approaches where a considerable performance gain can be achieved, has tended to have limited their use. More importantly, to the authors' knowledge, the necessary performance gain over existing approaches has not been demonstrated substantially enough to merit wider adoption. Given that most FPGA architectures have dedicated hardware based on conventional arithmetic, this somewhat skews the focus towards conventional two's complement-based processing. For this reason, much of the description and the examples in this text, have been restricted to two's complement.

3.3 Fixed-point and Floating-point

A widely used format for representing and storing numerical data in the binary number system, is the fixed-point format. In fixed-point arithmetic, an integer value x represented by the series of bits $x_{m+n-1}, x_{m+n-2}, \ldots, x_0$ is mapped in such a way that $x_{m+n-1}, x_{m+n-2}, \ldots, x_n$ represents the integer part of the number and $x_{n-1}, x_{n-2}, \ldots, x_0$ represents the fractional part of the number. This is the interpretation placed on the number system by the user and generally in DSP systems users represent input data, say $x(n)$, and output data, $y(n)$, as integer values and coefficient word values as fractional so as to maintain the best dynamic range in the internal calculations.

The key issue when choosing a fixed-point representation is to best use the dynamic range in the computation. Scaling can be applied to cover the worst-case scenario, but this will usually result in poor dynamic range. Adjusting to get the best usage of the dynamic range usually means that overflow will occur in some cases and additional circuitry has to be implemented to cope with this condition; this is particualrly problematic in two's complement as overflow results in an 'overflowed' value of completely different sign to the previous value. This can be avoided by introducing saturation circuitry to preserve the worst-case negative or positive overflow, but this has a non-linear impact on performance and needs further investigation.

This issue is usually catered for in the high-level modeling stage using tools such as those from Matlab® or Labview. These tools allow the high-level models to be developed in a floating-point representation and then be translated into a fixed-point realization. At this point, any overflow problems can be investigated. A solution may have implications for the FPGA implementation aspects as the timing problems may now exist as a result of the additional circuitry. This may seem to be more trouble than it is worth, but fixed-point implementations are particularly attractive for FPGA implementations (and for some DSP microprocessor implementations), as word size translates directly to silicon area. Moreover, a number of optimizations are available that make fixed-point extremely attractive; these are explored in later chapters.

3.3.1 Floating-point Representations

Floating-point representations provide a much more extensive means for providing real number representations and tend to be used extensively in scientific computation applications, but also

S	Exp	Fraction
1	10001001	10000111010101010000000

31 30 (8-bits) 22 (23-bits) 0 bit index
bias = +127

(a) Single precision

S	Exp	Fraction
1	10000001001	1000011101010101000000000000000000000000000000000000

63 62 (11-bits) 52 51 (52-bits) 0 bit index
bias = +1023

(b) Double precision

Figure 3.2 Floating-point representations

increasingly, in DSP applications. In floating-point, the aim is to represent the real number using a *sign (S)*, *exponent (Exp)* and *mantissa* (or fraction) as shown in Figure 3.2. The most widely used form of floating-point is IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754). It specifies four formats:

- single-precision (32-bit)
- double-precision (64-bit)
- single-extended precision
- double-extended precision

The single-precision is a 32-bit representation where one bit is used for the sign (S), 8 bits for the exponent (Exp) and 23 bits for the mantissa (M). This is illustrated in Figure 3.2(a). This allows representation of the number N where N is created by $2^{Exp-127} \times M$ as the exponent is represented as unsigned, giving a single-extended number of approximately $\pm\ 10^{38.53}$. The double precision is a simple extension of the concept to 64 bits, allowing a range of $\pm\ 10^{308.25}$.

The simple example below shows how a real number, -1082.5674 is converted into IEEE 754 floating-point representation.

It can be determined that $S = 1$ as the number is negative.
The number (1082) is converted to binary by successive division (Omondi 1994), giving 10000111010.
The fractional part (0.65625) is computed in the same way as above, giving 10101.
The parts are combined to give the value 10000111010.10101.
The radix point is moved left, to leave a single 1 on the left, giving $1.000011101010101 \times 2^{10}$.
Filling with 0s to get the 23-bit mantissa gives the value 10000111010101010000000.
The exponent is 10 and with the 32-bit IEEE 754 format bias of 127, giving 137 which is given as 10001001 in binary.

Putting all this together gives the binary pattern of Figure 3.2(a) for single precision. The same process is applied to double precision given in Figure 3.2(b), with the main difference being the offset added to the exponent, and the addition zero padding for the mantissa.

3.4 Arithmetic Operations

This section looks at the implementation of various arithmetic functions, including addition and multiplication and also division and square root. As the emphasis is on FPGA implementation which comprises on-board adders and multipliers, the book concentrates on using these constructions, particularly fixed-point realizations. A brief description of a floating-point adder is given in the following section.

3.4.1 Adders and Subtracters

Addition is a key operation in itself, but also forms the basic unit of multiplication which is, in effect, a series of shifted additions. The basic addition function is given in Table 3.2 and the resulting implementation in Figure 3.3(a). This form comes directly from solving the 1-bit adder truth table leading to Equations (3.6) and (3.7) and the logic gate implementation of Figure 3.3(a).

$$S_i = A_i \oplus B_i \oplus C_{i-1} \tag{3.6}$$

$$C_i = A_i \cdot B_i + A_i \cdot C_{i-1} + B_i \cdot C_{i-1} \tag{3.7}$$

The truth table can also be interpreted as follows: when $A_i = B_i$, then $C_i = B_i$ and $S_i = C_{i-1}$; when $A_i = \overline{B}_i$, then $C_i = C_{i-1}$ and $S_i = \overline{C}_{i-1}$. This implies a multiplexer for the generation of the carry and, by cleverly using $A_i \oplus B_i$ (already generated in order to develop the sum value, S_i), very little additional cost is required. This is the preferred construction for FPGA vendors as indicated by the partition of the adder cell in Figure 3.3(b). By providing a dedicated *EXOR* and *MUX* logic, the adder cell can then be built by using the LUT to generate the additional *EXOR* function.

There has been a considerable detailed investigation of adder structures in the computer arithmetic field as the saving of a few hundred picoseconds has considerable performance impact. A wide range of adder structures have been developed including carry-ripple or ripple-carry, carry lookahead, carry-save, carry skip, conditional sum, to name but a few (Omondi 1994). The ripple-carry or carry-ripple adder is highlighted in Figure 3.4 which gives a 4-bit adder implementation where each of the cells are defined logic represented by Equations (3.6) and (3.7).

Table 3.2 Truth table for a 1-bit adder

	Inputs			Outputs	
	A	B	C_i	S_o	C_o
C_o = A or B	0	0	0	0	0
	0	0	1	1	0
	0	1	0	1	0
$C_o = C_i$	0	1	1	0	1
	1	0	0	1	0
	1	0	1	0	1
C_o = A or B	1	1	0	0	1
	1	1	1	1	1

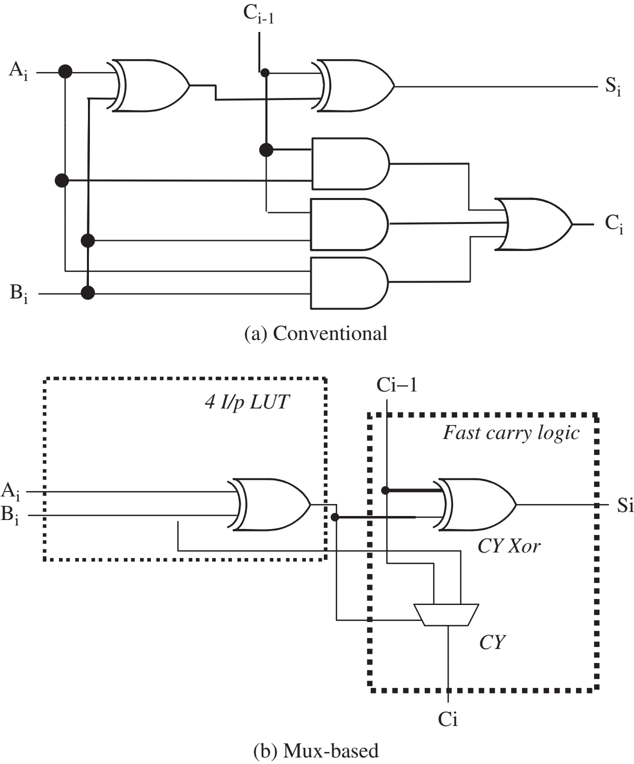


Figure 3.3 1-bit adder structures

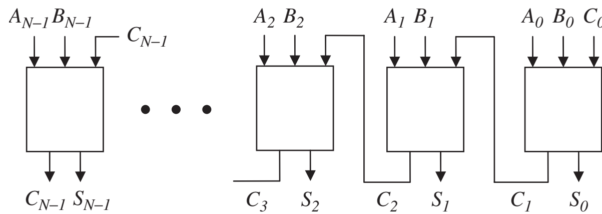


Figure 3.4 N-bit adder structure

To a large extent, the variety of different adder structures trade off gate complexity with system regularity, as many of the techniques end up with structures that are much less regular. The aim of much of the research which took place in the 1970s and 1980s, was to develop higher-speed structures where transistor switching speed was the dominant feature. However, the analysis in the introduction to the book, indicates the key importance of interconnect, and somewhat reduces the impact of using specialist adder structures. Another critical consideration for FPGAs is the importance of being able to scale adder word sizes with application need, and in doing so, offer a linear scale in terms of performance reduction.

For this reason, the ripple-carry adder has great appeal in FPGAs and is offered in many of the FPGA structures as a dedicated resource (see Chapter 5).

3.4.2 Multipliers

Multiplication can be simply performed through a series of additions. Consider the example below, which illustrates how the the simple multiplication of 5 by 11 is carried out in binary.

```
 5 = 00101  multiplicand
11 = 01011  multiplier
     -----
     00101
    00101
   00000
  00101
 00000
     --
55 = 000110111
```

The usual terminology in computer arithmetic is to align the data in a vertical line and shift right rather than shift left, as shown below. However, rather than perform one single addition at the end to add up all the *multiples*, each multiple is added to an ongoing product called a *partial product*. This means that every step in the computation equates to the generation of the multiples using an *and* function or gate and the use of an adder to compute the partial product.

```
 5 = 00101  multiplicand
11 = 01011  multiplier
     -----
     00000  initial partial product
     00101  add 1st multiple
     -----
     00101
     000101  shift right
     00101   add 2nd multiple
     -----
     001111
     0001111  shift right
     00000      add 3rd multiple
     -----
     0001111
     00001111  shift right
     00101       add 4th multiple
     -----
     00110111
     00001111  shift right
     00000        add 5th multiple
     -----
55 = 000110111
```

The repetitive nature of the computation implies a serial processor unit for each stage, comprising an array of *and* gates to create the product terms and an adder to perform the stage-wise addition. With improvements in silicon technology though, parallel multipliers have now become the norm, which means that multiple adder circuits can be used. However, if the adder structures of Figure 3.4 were used, this would result in a very slow multiplier circuit. Use of fast adders would improve the speed, but would result in increased hardware cost. For this reason, the carry-save adder structure of Figure 3.5 is used to generate these additions as indicated in the carry-save array multiplier (Figure 3.6). The carry-save adder is as fast as the individual cell namely 3 gate delays, as given in Figure 3.3. This arrangement allows a final sum and carry to be quickly generated; a fast adder given as CPA, is then used to produce the final sum.

Even though each addition stage is reduced to two or three gate delays, the speed of the multiplier is then determined by the number of stages. As the word size m grows, the number of stages is then given as $m - 2$. This limitation is overcome in a class of multipliers known as Wallace tree multipliers, which allows the addition steps to be perform in parallel. An example is shown in Figure 3.7. As the function of the carry-save adder is to compress three words to two words, this means that if n is the input wordlength, then after each stage, the n words are represented as $3k+1$

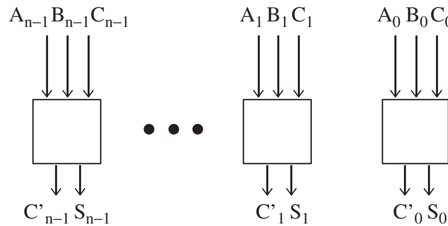


Figure 3.5 Carry-save adder

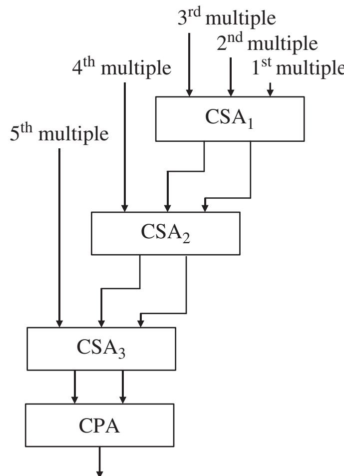


Figure 3.6 Carry-save array multiplier

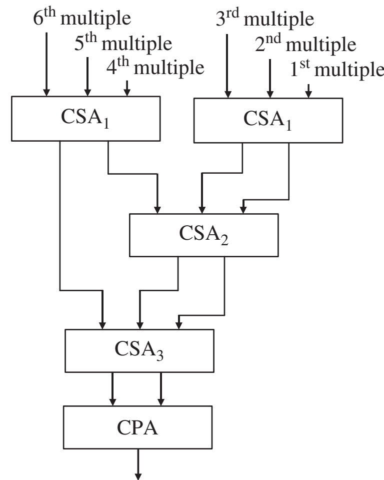


Figure 3.7 Wallace tree multiplier

where $0 \leq l \leq 2$. This means that the final sum and carry values are produced after $\log_{1.5} n$ rather than n-1 stages as with the carry-save array multiplier.

3.4.3 Division

Division may be thought of as the inverse process of multiplication, but it differs in several aspects that make it a much more complicated function. There are a number of ways of performing division, including *recurrence division* and *division by functional iteration*. Algorithms for division and square root have been a major research area in the field of computer arithmetic since the 1950s. The methods can be divided into two main classes, namely *digit-by-digit* methods and *convergence* methods. The digit-by-digit methods, also known as *direct* methods, are somewhat analogous to the *pen and paper* method of computing quotients and square roots. The results are computed on a digit-by-digit basis, *most significant digit* (msd) first. The convergence methods, which include the Newton–Raphson algorithm and the Taylor series expansion, require the repeated updating of an approximation to the correct result.

Recurrence Division

Digit recurrence algorithms are well-accepted subtractive methods which calculate quotients one digit per iteration. They are analogous to the pencil and paper method in that they start with the msbs and work toward the lsbs. The partial remainder is initialized to the dividend, then on each iteration, a digit of the quotient is selected according to the partial remainder. The quotient digit is multiplied by the divisor and then subtracted from the partial remainder. If negative, the restoring version of the recurrence divider restores the partial remainder to the previous value, i.e. the results of one subtraction (comparison) determine the next division iteration of the algorithm, which requires the selection of quotient bits from a digit set. Therefore, a choice of quotient bits needs to be made at each iteration by *trial and error*. This is not the case with multiplication, as the partial products may be generated in parallel, and then summed at the end. These factors make division a more complicated algorithm to implement than multiplication and addition.

When dividing two n-bit numbers this method may require up to $2n + 1$ additions. This can be reduced by employing the *non-restoring recurrence algorithm* in which the digits of the partial remainder are allowed to take negative and positive values; this reduces the number of additions to n. The most popular recurrence division method is an algorithm known as the SRT division algorithm which was named after the initials of the three researchers who independently developed it, Sweeney, Robertson and Tocher (Robertson 1958, Tocher 1958).

The recurrence methods offer simple iterations and smaller designs, however, they also suffer from high latencies and converge linearly to the quotient. The number of bits retired at each iteration depends on the radix of the arithmetic being used. Larger radices may reduce the number of iterations required, but will increase the time for each iteration. This is because the complexity of the selection of quotient bits grows exponentially as the radix increases, to the point that lookup tables (LUTs) are often required. Therefore, a trade-off is needed between the radix and the complexity; as a result the radix is usually limited to 2 or 4.

Division by Functional Iteration

The digit recurrence algorithms mentioned in the previous section, retire a fixed number of bits at each iteration, using only shift and add operations. Functional iterative algorithms on the other hand employ multiplication as the fundamental operation and produce at least double the number of correct bits with each iteration (Flynn 1970, Ito *et al.* 1995, Obermann and Flynn 1997, Oklobdzija and Ercegovac 1982). This is an important factor as there may be as many as three multiplications in each iteration. However, with the advantage of at least quadratic convergence, a 53-bit quotient can be achieved in 6 iterations, as shown in Figure 3.8.

3.4.4 Square Root

Methods for performing the square root operation are similiar to those for performing division. They fall broadly into the two categogies, digit recurrence methods and methods based on convergence techniques. The following sections give a brief overview of each.

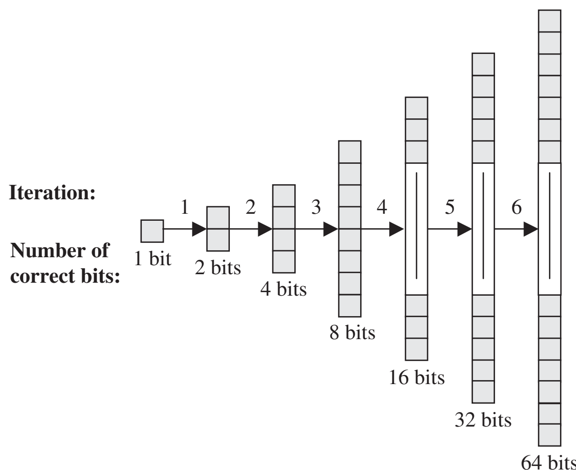


Figure 3.8 Quadratic convergence

Digit Recurrence Square Root

Digit recurrence methods can either be restoring or non-restoring techniques, both of which operate *msd* first. The algorithm is subtractive and after each iteration, the resulting bit is set to 0 if a negative value is found, and then the original remainder is 'restored' as the new remainder. If digit is positive, a 1 is set and the new remainder is used. The 'non-restoring' algorithm allows the negative value to persist and then performs a compensative addition operation in the next iteration. The overall process between the square root and division algorithms is very similar and, as such, there have been a number of implementations of systolic arrays designed to perform both arithmetic functions (Ercegovac and Lang 1991, Heron and Woods 1999).

The algorithms mentioned have limited performance due to the dependence of the iterations and the propagated carries along each row. The full values need to be calculated at each stage to enable a correct comparison and decision to be made. The SRT algorithm is a class of non-restoring digit-by-digit algorithms in which the digit can assume both positive and negative nonzero values. It requires the use of a redundant number schemes (Avizienis 1961), thereby allowing digits to take the values of 0, -1 or 1. The most important feature of the SRT method is that the algorithm allows each iteration to be performed without full precision comparisons at each iteration, thus giving higher performance.

Consider a value R for which the algorithm is trying to find the square root, and S_i is the partial square root obtained after i iterations. The scaled remainder at the ith step is:

$$Z_i = 2^i(R - S_i^2) \tag{3.8}$$

where $1/4 \geq R > 1$ and hence $1/2 \geq S < 1$. From this, a recurrence relation based on previous remainder calculations can be derived as (McQuillan *et al.* 1993):

$$Z_i = 2Z_{i-1} - s_i(2S_{i-1} + s_i 2^{-i}) \quad i = 2, 3, 4 \ldots \tag{3.9}$$

where, s_i is the root digit for iteration $i - 1$.

Typically, the initial value for Z_0 will be set to R, while the initial estimate of the square root, S_1 is set to 0.5, (due to the initial boundaries placed on R).

Higher-radix square root algorithms exist (Ciminiera and Montuschi 1990, Cortadella and Lang 1994, Lang and Montuschi 1992). However, for most algorithms with a radix greater than 2, there is a need to provide an initial estimate to the square root from a LUT. This relates to the following section.

Square Root by Functional Iteration

As with the convergence division in Section 3.4.3, square root calculation can be performed using functional iteration. They can be additive or multiplicative. If additive, then each iteration is based on addition and will retire the same number of bits with each iteration. In other words, they converge linearly to the solution. An example of such an algorithm is CORDIC, one use of which has been in performing Givens rotations for matrix triangularization (Hamill *et al.* 2000). Mulitplicative algorithms offer an interesting alternative as they double the precision of the result with each iteration, that is, they converge quadratically to the result. However, they have the disdvantage of the increased computational complexity due to the multiplications within each iterative step.

Similarly to the division methods, the square root can be estimated using Newton–Raphson or series convergence algorithms. For the Newtown–Raphson method an iterative algoritm can be found by using:

$$x_{i+1} = X_i - \frac{f(x_i)}{f'(x_i)} \tag{3.10}$$

and choosing $f(x)$ that has a root at the solution. One possible choice is $f(x) = x^2 - b$ which leads to the following iterative algorithm:

$$x_{i+1} = 1/2\left(X_i - \frac{b}{x_i}\right) \tag{3.11}$$

This had the disadvantage of requiring division. An alternative method would be to aim to drive the algorithm towards calculating the reciprocal of the square root, that is $1/x^2$. For this, $f(x) = 1/x^2 - b$ is used which leads to the following iterative algorithm:

$$x_{i+1} = \frac{x_i}{2}\left(3 - bx_i^2\right) \tag{3.12}$$

Once solved, the square root can then be found by multiplying the result by the original value X, that is, $1/\sqrt{X} \times X = \sqrt{X}$.

Another method for implementing the square root function is to use series convergence (Soderquist and Leeser 1995), Goldschmidt's algorithm) which produces equations similar to those equations for division (Even *et al*. 2003).

The aim of this algorithm is to compute successive iterations to drive one value to 1 while driving the other value to the desired result. To calculate the square root of a value a, for each iteration:

$$x_{i+1} = x_i \times r_i^2 \tag{3.13}$$

$$y_{i+1} = y_i \times r_i \tag{3.14}$$

where we let, $x_0 = y_0 = a$. Then by letting:

$$r_i = \frac{3 - y_i}{2} \tag{3.15}$$

$x_i \rightarrow 1$ and consequently $y_i \rightarrow \sqrt{a}$. In other words, with each iteration x is driven closer to 1 while, y is driven closer to $\sqrt{a}$.

As with the other convergence examples, the algorithm benefits from using an initial estimate of $1/\sqrt{a}$ to prescale the initial values of x_0 and y_0.

In all of the examples given for both the division and square root convergence algorithms, vast improvements in performance can be obtained by using a LUT to provide an initial estimate to the desired solution. This is covered in the following section.

Initial Approximation Techniques

The number of iterations for convergence algorithms can be vastly reduced by providing an initial approximation to the result read from a LUT. For example, the simplest way of forming the approximation R_0 to the reciprocal of the divisor D, is to read an approximation to $1/D$ directly out of a LUT. The first m bits of the n-bit input value D are used to address the table entry of p bits holding an approximation to the reciprocal. The value held by the table is determined by considering the maximum and minimum errors caused by truncating D from n to m bits.

The access time to a LUT is relatively small so it provides a quick evaluation of the first number of bits to a solution. However, as the size of the input value addressing the LUT increases, the size

Table 3.3 Precision of approximations for example values of g and m

Address bits	Guard bits g	Out bits	Precision at least
m	0	m	$m + 0.415$bits
m	1	$m + 1$	$m + 0.678$bits
m	2	$m + 2$	$m + 0.830$bits
m	3	$m + 3$	$m + 0.912$bits

of the table grows exponentially. For a table addressed by m bits and outputting p bits the table size will have $2m$ entries of width p bits. Therefore, the size of the LUT soon becomes very large and will have slower access times.

A combination of p and m can be chosen to achieve the required accuracy for the approximation, with the smallest possible table. By denoting the number of bits that p is larger than m as the number of guard bits g, the total error E_{total} may be expressed as (Sarma and Matula 1993):

$$E_{\text{total}} = 2^{m+1}\left(\frac{1}{2^{g+1}}\right) \tag{3.16}$$

Table 3.3 shows the precision of approximations for example values of g and m. These results are useful in determining whether adding a few guard bits might provide sufficient additional accuracy in place of the more costly step in increasing m to $m + 1$ which more than doubles the table size.

Another simple approximation technique is known as ROM interpolation. Rather than just truncating the value held in memory after the mth bit, the first unseen bit $(m + 1)$ is set to 1, and all bits less significant than it, are set to 0 (Fowler and Smith 1989). This has the effect of averaging the error. The resulting approximate is then rounded back to the *lsb* of the table entry by adding a 1 to the bit location just past the output width of the table. The advantage with this technique is its simplicity. However, it would not be practical for large initial approximations as there is no attempt to reduce the table size.

There are techniques for table compression, such as bipartite tables, which use two or more LUTs and then add the output values to determine the approximation. To approximate a reciprocal function using bipartite tables the input operand is divided into three parts as shown in Figure 3.9.

The $(n_0 + n_1)$ bits provide the address for the first LUT, giving the coefficient a_0 of length p_0 bits. The sections d_0 and d_2, equating to $(n_0 + n_2)$ bits provide addresses for the second LUT, giving the second coefficient a_1 of length p_1 bits. The outputs from the tables are added together to approximate the reciprocal, R_0, using a two-term Taylor series expansion. The objective is to use the first $(n_0 + n_1)$ *msb*s to provide the lookup for the first table which holds coefficients based on the values given added with the mid-value of the range of values for d_2. The calculation of the second coefficient is based on the value from sections d_0 and d_2 summed with the mid-value of the range of values for d_1. This technique forms a method of averaging so that the errors caused by truncation are reduced. The coefficients for the reciprocal approximation take the form:

$$a_0(d_0, d_1) = f(d_0 + d_1 + \delta_2) = \frac{1}{d_0 + d_1 + \delta_2} \tag{3.17}$$

$$a_0(d_0, d_1) = f'(d_0 + d_1 + \delta_2)(d_2 = \delta_2) = \frac{\delta_2 - d_2}{(d_0 + \delta_1 + \delta_2)^2} \tag{3.18}$$

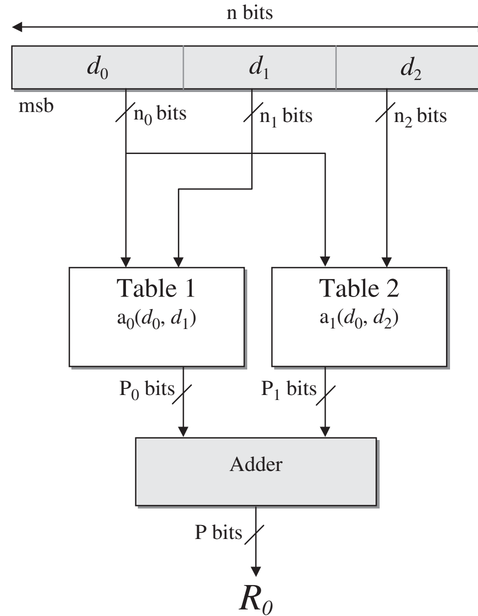


Figure 3.9 Block diagram for the bipartite approximation method (Schulte *et al.* 1997)

where, δ_1 and δ_2 are constants exactly halfway between the minimum and maximum values for d_1 and d_2 respectively.

The benefit is that the two small LUTs will have less area than the one large LUT for the same accuracy, even when the size of the addition is considered. Techniques to simplify the bipartite approximation method also exist. One method, (Sarma and Matula 1995), eliminates the addition by using each of the two LUTs to store the positive and negative portions of a redundant binary reciprocal value. These are 'fused' with slight recoding to round off a couple of low-order bits to obtain the required precision of the least significant bit. With little extra logic this recoding can convert the redundant binary values into Booth encoded operands suitable for input into a Booth encoded multiplier.

3.5 Fixed-point versus Floating-point

If the natural assumption is that 'more accurate is always best', then there appears no choice in determining the number representation as floating-point will be chosen. However, the area cost of floating-point, particularly for FPGA implementations, is prohibitive, with area costs for some DSP applications being quoted as much as 10 times larger than fixed-point (Lightbody *et al.* 2007). Take, for example, the floating-point adder given in Figure 3.10 which is derived from Pillai *et al.* (2001) and is typical of adder implementations. A first examination shows that there is a number of adders/subtracters, as well as barrel shifters and other control logic. This additional logic is needed to perform the various normalization steps for the adder implementation. This is highlighted in Table 3.4 which shows the various functionality required, namely circuit A, circuit B and circuit C, i.e. bypass (illustrated in dashed lines in Figure 3.10). The table gives a simplified interpretation

Table 3.4 Simplified version of exponent criteria for floating-point adder operation (Pillai *et al*. 2001)

Exponent criteria	Circuitry activated
$\|e1 - e2\| > p$	C
$\|e1 - e2\| \leq p$ and subtraction	B
$\|e1 - e2\| \leq p$ and addition	A

for what circuitry is needed for the various conditions of the exponent values of the two numbers namely $e1$ and $e2$ and the significance width, p. Please note this does not consider special cases such as $0 \pm operand$ or $\infty \pm operand$ as the idea of the example has just been to give some idea of complexity and to indicate why the cost of floating-point hardware is much larger than that of fixed-point.

The area comparison for floating-point is additionally complicated as the relationship between multiplier and adder area is now changed. In fixed-point, multipliers are generally viewed to be N times bigger than adders where N is the wordlength. However, in floating-point, the area of floating-point adders is comparable to that of floating-point multipliers which corrupts the assumption at the algorithmic stages to reduce number of multiplications in favour of additions. Table 3.5 gives some figures taken from Lightbody *et al*. (2007) which gives area and speed figures for floating-point addition and multiplication implemented in a Xilinx Virtex 4 FPGA technology. Figures are also

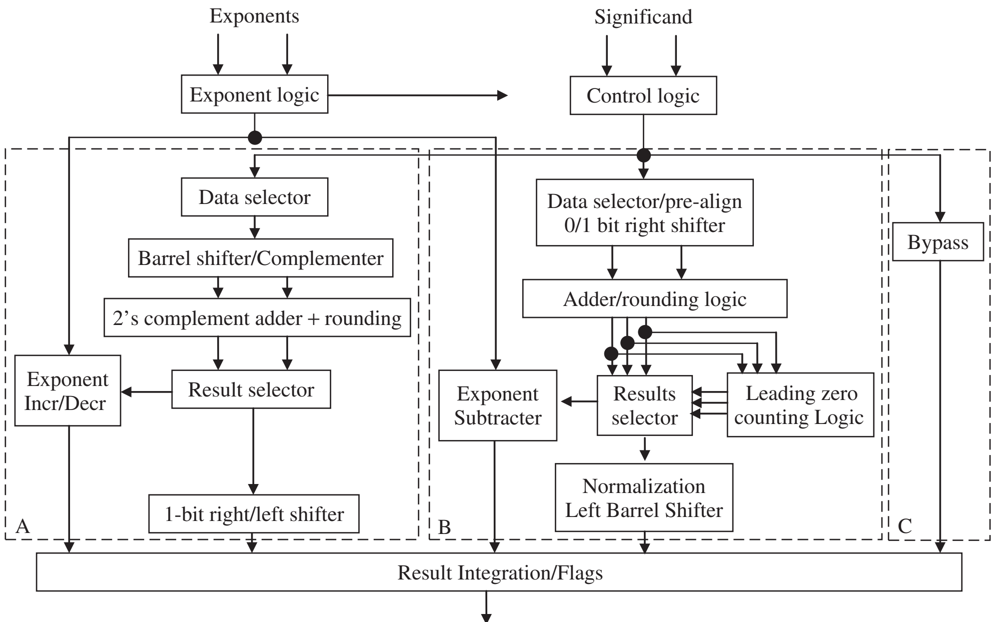


Figure 3.10 Triple data path floating-point adder block diagram

Table 3.5 Area and speed figures for various floating-point operators implemented using Xilinx Virtex 4 FPGA technology

Function	DSP48	LUT	Flip-flops	Speed (MHz)
Multiplier	4	799	347	141.4
Adder	×	620	343	208.2
Reciprocal	4	745	266	116.5

Table 3.6 Typical wordlengths

Application	Word sizes (bits)
Control systems	4–10
Speech	8–13
Audio	16–24
Video	8–10

included for a special case of floating-point division, namely a reciprocal function. The values are based on the variable precision floating-point modules available from North Eastern University.

The decision though, is more complex than just a simple area and speed comparison, and should be judged on the actual application requirements. For example, many applications vary in terms of the data word sizes and the resulting accuracy. Applications can require different input wordlengths, as illustrated in Table (3.6) and can vary in terms of their sensitivity to errors created as result of limited, internal wordlength. Obviously, smaller input wordlengths will have smaller internal accuracy requirements, but the perception of the application will also play a major part in determining the internal wordlength requirements. The eye is tolerant of wordlength limitations in images, particularly if they appear as distortion at high frequencies, whereas the ear is particularly intolerant to distortion and noise at any frequency, but specifically high frequency. Therefore cruder truncation may be possible with some image processing applications, but less so in audio applications.

Table (3.7) gives some estimation of the dynamic range capabilities of some fixed-point representations. It is clear that, depending on the internal computations being performed, many DSP applications can give acceptable *signal-to-noise ratios* (SNRs) with limited wordlengths, say, 12–16 bits. Given the performance gain of fixed-point over floating point in FPGAs, this has meant that fixed-point realizations have dominated, but the choice will also depend on application input and output wordlengths, required SNR, internal computational complexity and the nature

Table 3.7 Fixed wordlength dynamic range

Wordlength (bits)	Wordlength range	Dynamic range dB
8	-127 to $+127$	$20\log 2^8 \simeq 48$
16	-32768 to $+32767$	$20\log 2^{16} \simeq 96$
24	-8388608 to $+8388607$	$20\log 2^{24} \simeq 154$

of computation being performed, i.e. whether specialist operations such as matrix inversions or iterative computations, are required.

A considerable body of work has been dedicated to reduce the number precision to best match the performance requirements. In Constantinides *et al*. (2004), the author looks to derive accurate bit approximations for internal wordlengths by considering the impact on design quality. A floating-point design flow is presented in (Fang *et al*. 2002) which takes an algorithmic input, and generates floating-point hardware by performing bit width optimization, with a cost function related to hardware, but also to power consumption. This activity is usually performed manually by the designer, using suitable fixed-point libraries in tools such as Matlab® or Labview, as suggested earlier.

3.6 Conclusions

The chapter has given a brief grounding in computer arithmetic basics and given some idea of the hardware needed to implement basic computer arithmetic functions and some more complex functions such as division and square root. In addition, the chapter has highted some critical aspects of arithmetic representations and the implications that choice of either fixed- or floating-point arithmetic can have in terms of hardware implementation, particularly given the current FPGA support for floating-point. It clearly demonstrates that FPGA technology is currently very appropriate for fixed-point implementation, but much less so for floating-point arithmetic.

References

Andrews M (1986) A systolic sbnr adaptive signal processor. *IEEE Trans. on Circuits and Systems* **33**(2), 230–238.

Avizienis A (1961) Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers* **EC-10**, 389–400.

Chandrakasan A and Brodersen R (1996) *Low Power Digital Design*. Kluwer, Dordrecht.

Ciminiera L and Montuschi P (1990) Higher radix square rooting. *IEEE Trans. Comput.* **39**(10), 1220–1231.

Collange S, Detrey J and de Dinechin F (2006) Floating point or lns: choosing the right arithmetic on an aapplication basis. *Proc. 9th EUROMICRO Conf. on Digital System Design*, pp. 197–203.

Constantinides G, Cheung PYK and Luk W (2004) *Synthesis and Optimization of DSP Algorithms*. Kluwer, Dordrecht.

Cortadella J and Lang T (1994) High-radix division and square-root with speculation. *IEEE Trans. Comput.* **43**(8), 919–931.

Ercegovac MD and Lang T (1990) Redundant and on-line cordic: Application to matrix triangularization. **39**(6), 725–740.

Ercegovac MD and Lang T (1991) Module to perform multiplication, division, and square root in systolic arrays for matrix computations. *J. Parallel Distrib. Comput.* **11**(3), 212–221.

Even G, Seidel PM and Ferguson W (2003) A parametric error analysis of Goldschmidt's division algorithm, pp. 165–171.

Fang F, Chen T and Rutebnbar RA (2002) Floating-point bit-width optimisation for low-power signal processing applications *Proc. Int. Conf. on Acoustics, Speech and Signal Proc.*, vol. 3, pp. 3208–3211.

Flynn M (1970) On division by functional iteration. *IEEE Trans. on Computing* **C-19**(8), 702–706.

Fowler D L and Smith JE (1989) High speed implementation of division by reciprocal approximation *IEEE Symp. on Computer Arithmetic*, pp. 60–67.

Hamill R, McCanny J and Walke R (2000) Online CORDIC algorithm and vlsi architecture for implementing qr-array processors. *IEEE Trans. on Signal Processing* **48**(2), 592–598.

Heron JP and Woods RF (1999) Accelerating run-time reconfiguration on FCCMS. *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 260–261.

Ito M, Takagi N and Yajima S (1995) Efficient initial approximation and fast converging methods for division and square root. *Proc. 12th IEEE Symp. on Comp. Arith.*, pp. 2–9.

Knowles S, Woods R, McWhirter J and McCanny J (1989) Bit-level systolic architectures for high performance IIR filtering. *Journal of VLSI Signal Processing* **1**(1), 9–24.

Lang T and Montuschi P (1992) Higher radix square root with prescaling. *IEEE Trans. Comput.* **41**(8), 996–1009.

Lightbody G, Woods R and Francey J (2007) Soft IP core implementation of recursive least squares filter using only multiplicative and additive operators *Proc. Int. Conf. on Field Programmable Logic*, pp. 597–600.

Ma J, Deprettere EF and Parhi K (1997) Pipelined cordic based QRD-RLS adaptive filtering using matrix lookahead *Proc. IEEE Intl. Workshop-SiPS*, pp. 131–140.

McQuillan SE, McCanny J and Hamill R (1993) New algorithms and VLSI architectures for SRT division and square root. *Proc. IEEE Symp. Computer Arithmetic*, pp. 80–86.

Muller JM (2005) *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston.

Obermann SF and Flynn MJ (1997) Division algorithms and implementations. *IEEE Trans. Comput.* **C-46**(8), 833–854.

Oklobdzija V and Ercegovac M (1982) On division by functional iteration. *IEEE Trans. Comput.* **C-31**(1), 70–75.

Omondi AR 1994 *Computer Arithmetic Systems*. Prentice Hall, New York.

Pillai RVK, Al-Khalili D, Al-Khalili AJ and Shah SYA (2001) A low power approach to floating point adder design for DSP applications. *Journal of VLSI Signal Proc.* **27**, 195–213.

Robertson J (1958) A new class of division methods. *IRE Trans. on Electronic Computing* **EC-7**, 218–222.

Sarma DD and Matula DW (1993) Measuring the accuracy of ROM reciprocal tables *Proc. IEEE Symp. on Computer Arithmetic*, pp. 95–102.

Sarma DD and Matula DW (1995) Faithful bipartite rom reciprocal tables. *Proc. IEEE 12th Symposium on Computer Arithmetic*, pp. 17–28.

Schulte MJ, Stine JE and Wires KE (1997) High-speed reciprocal approximations. *Proc. 31st Asilomar Conference on Signals, Systems and Computers*, pp. 1183–1187.

Sklansky J (1960) Conditional sum addition logic. *IRE Trans. Elect. Comp.* **EC-9**(6), 226–231.

Soderquist P and Leeser M (1995) An area/performance comparison of subtractive and multiplicative divide/square root implementations *Proc. IEEE Symp. on Computer Arithmetic*.

Soderstrand MA, Jenkins WK, Jullien GA and Taylor FA (1986) *Residue Number Systems Arithmetic: Modern Applications in Digital Signal Processing*. IEEE Press.

Takagi N, Asada T and Yajima S (1991) Redundant CORDIC methods with a constant scale factor for sine and cosine computation. **40**, 989–995.

Tocher K (1958) Techniques of multiplication and division for automatic binary computers. *Quart. J. Mech. Appl. Math.* **XI**(3), 364–384.

Volder JE (1959) The CORDIC trigonometric computing technique. **EC-8**, 330–334.

Walther JS (1971) A unified algorithm for elementary functions. *Spring Joint Comput. Conf.*, pp. 379–385.

4

Technology Review

4.1 Introduction

The technology used for DSP implementation is very strongly linked with the astonishing developments in silicon technology. As was highlighted in the introduction to this book, the availability of a transistor which has continually decreased in cost, has been the major driving force in creating new markets and has overseen the development of a number of DSP technologies. Silicon technology has not only offered an increasingly cheaper platform, but has also offered this at higher speeds and at a lower power cost. This has inspired a number of DSP-based markets, specifically mobile telephony and digital video products.

As Chapter 2 clearly indicated, there are numerous advantages of systems in the digital domain, specifically guaranteed accuracy, essentially perfect reproducibility and better ageing; these developments are seen as key to the continued realization of future systems. The earliest DSP filter circuits were pioneered by Leland B. Jackson and colleagues at Bell laboratories (Jackson 1970) in the late 1960s and early 1970s. At that time, the main aim was to create silicon chips for performing basic filtering functions such as FIR and IIR filtering. A key aspect was the observation that the binary operation of the transistor, could be well matched, to creating the necessary digital operation required in DSP systems.

From these early days, a number of technologies have emerged; these range from simple microcontrollers where the performance requirement, typically *sampling rate*, are in the moderate *kHz* range, right through to dedicated DSP SoCs that give performance nearing the TeraOPS range. This *processor style* architecture has been exploited in various forms ranging from single to multi-core processor implementations, dedicated DSP microprocessors where hardware has been included to allow specific DSP functionality to be realized efficiently, and also reconfigurable DSP processor architectures. Another concept is the development of application specific instruction processors (ASIPs) that have been developed for specific application domains. The authors would argue that the main criteria in DSP system implementation is in terms of the *circuit architecture* employed. Generally speaking, the hardware resources and how they are interconnected have a major part to play in the performance of the resulting DSP system. FPGAs allow this architecture to be created to best match the algorithmic requirements, but this comes at increased design cost. It is interesting to compare the various approaches, and thus the chapter aims to give an overview of the various technologies, available for implementing DSP systems, using relevant examples where applicable. In addition, the technologies are also compared and contrasted. As Chapter 5 is dedicated to the variety of FPGA architectures, the FPGA material is only covered briefly here. The major themes considered in the description include the level of programmability, the programming

FPGA-based Implementation of Signal Processing Systems R. Woods, J. McAllister, G. Lightbody and Y. Yi
© 2008 John Wiley & Sons, Ltd

environment (including tools, compilers and frameworks), the scope for optimization of specifically DSP functionality on the required platform, and quality of the resulting designs in terms of area, speed, throughput, power and even robustness. The chapter is broken down as follows. Section 4.2 outlines some further thoughts on circuit architecture, giving some insight toward the performance limitations of the technologies and also, comments on the importance of programmability. In Section 4.3, the functional requirements of DSP systems are examined, highlighting issues such as computational complexity, parallelism, data independence and arithmetic advantages. Section 4.4 outlines the processor classification and is followed by a brief description of microprocessors in Section 4.5, and DSP processors in Section 4.6. A number of parallel machines are then described in Section 4.7 which include systolic array architectures, single instruction multiple data (SIMD) and multiple instruction multiple data (MIMD) along with some examples. For completeness, the ASIC and FPGA route is briefly reviewed in Section 4.8, but only briefly as this forms the major focus of the rest of the book. The final section gives some thoughts of how the various technologies compare and sets the scene for FPGAs in the next chapter.

4.2 Architecture and Programmability

In many processor-based systems, design simply represents the creation of the necessary high-level code with some thought given to the underlying technology architecture, in order to optimize code quality and thus improve performance. Crudely speaking though, performance is sacrificed to provide this level of programmability. Take, for example, the microprocessor architecture based on the Von Neumann sequential model. The underlying architecture is fixed and the maximum, achievable performance will be determined by efficiently scheduling the algorithmic requirements onto the inherently sequential, processing architecture. If the computation under consideration is highly parallel in nature (as is usually the case in DSP), then the resulting performance could be poor. If we were to take the other extreme and develop an SoC-based architecture to best match the computational complexity of the algorithm by developing the *right* level of parallelism needed (*if such a concept exists*), then the best performance should be achieved in terms of area, speed and power consumption. This requires the use of a number of design activities to ensure that hardware implementation metrics best match the performance criteria of the algorithm, strictly application and indeed, that the resulting design operates correctly.

To more fully understand this concept of generating a *circuit architecture*, consider ‘state-of-the-art’ in 1969. Hardware capability in terms of numbers of transistors was limited and thus highly valuable, so the processing in the filters described in (Jackson 1970), had to be done in a rather serial fashion. With current FPGAs, the technology provides hundreds of bit parallel multipliers, so therefore the arithmetic style and resulting performance is therefore quite different, implying a very different sort of architecture. The aim is thus to make best use of the available hardware against the performance criteria of the application. Whilst this latter approach of developing the hardware to match the performance needs is highly attractive, the architecture development presents a number of problems related to the very process of producing this architecture, namely design time, verification and test of the architecture in all its various modes, and all the issues associated with producing a *right first time* design. Whilst the performance of implementing these algorithms on a specific hardware platform can be compared in terms of metrics such as throughput rate, latency, circuit area, energy, power consumption, etc. one major theme that can also be used to differentiate these technologies is *programmability*, or strictly speaking, *ease of programmability*. As will become clear in the descriptive material in this section, DSP hardware architectures can range in their level of programmability. A simplest platform with a fixed hardware architecture can

then be easily programmed using a high-level software language as, given the fixed nature of the platform, efficient software compilers can be and indeed have been, developed to create the most efficient realizations. However, as the platform becomes more complex and flexible, the complexity and efficiency of these tools is compromised, as now special instructions have to be introduced to meet this functionality. The main aim of the compiler is to take source code that may not have been written for the specific hardware architecture, and identify how these special functions might be applied to improve performance. In a crude sense, we suggest that making the circuit architecture *programmable* achieves the best efficiency in terms of performance, but presents other issues with regard to evolution of the architecture either to meet small changes in applications requirements or relevance to similar applications. This highlights the importance of tools and design environments, which is described in Chapter 11.

ASIC is at the other end of the spectrum from a programmability point-of-view; here, the platform will have been largely developed to meet the needs of the system under consideration or some domain-specific, standardized application. For example, WCDMA-based mobile phones require specific standardized DSP functionality which can be met by developing a SoC platform comprising processors and dedicated hardware IP blocks. This is essential to meet the energy requirements for most mobile phone implementations. However, the silicon fabrication costs have now pushed ASIC implementation into a specialized domain and typically solutions in this domain, are either for high volume, or have specific domain requirements, e.g. ultra-low power as in low-power sensors.

The concept of developing the architecture with some level of *hardware* programmability and also *software* programmability, is met with the FPGA architecture. The FPGA architecture largely comprises logic elements, LUTs, memory, routing, configurable I/O and some dedicated hardware, and provides the ideal framework for achieving a high level of performance.

4.3 DSP Functionality Characteristics

Typically, DSP operations are characterized as being: computationally intensive; highly suited to implementation with parallel processors, exhibiting a high degree of parallelism, data independent and in some cases, having lower arithmetic requirements than other high-performance applications, e.g. scientific computing. It is important to understand these issues more fully in order to judge their impact for mapping DSP algorithms onto hardware platforms such as FPGAs.

Computational Complexity

DSP algorithms can be highly complex. For example, consider the N-tap FIR filter expression given in the previous chapter as Equation (2.11) and repeated here (Equation 4.1) for convenience.

$$y_n = \sum_{i=0}^{N-1} a_i x_{n-i} \tag{4.1}$$

In effect, this computation indicates that a_0 must be multiplied by x_n, followed by the multiplication of a_1 by x_{n-1} to which it must be added, and so on. Given that the tap size is N, this means that the computation requires N multiplications followed by $N - 1$ additions in order to compute y_n as shown below

$$y_n = a_0 x_n + a_1 x_{n-1} + a_2 x_{n-2} + \ldots + a_{N-1} x_{n-N+1} \tag{4.2}$$

Given that another computation will start on the arrival of next sample, namely x_{n+1}, this defines the computations required per cycle, namely $2N$ operations per sample or two operations per tap. If a processor implementation is targeted, then this requires, say, a loading of the data every cycle which would need two or three cycles (to load data and coefficients) and to store the accumulating sum. This could mean an additional three operations per cycle, resulting in six operations per tap, or overall, $6N$ operations per sample. For an audio application with a sampling rate of 44.2 kHz, a 128-tap filter will require 33.9 megasamples/s (MSPS) which may seem realistic for some technologies, but when you consider image processing rates of 13.5 MHz, these computational rates quickly explode, resulting a computation rate of 10 gigasamples/s (GSPS). In addition, this may only be one function within the system and thus represent only a small proportion of the total processing required.

For a processor implementation, the designer will determine if the hardware can meet the throughput requirements by dividing the clock speed of the processor by the number of operations that need to be performed each cycle, as outlined above. This can give a poor return in performance, as if N is large, there will be a large disparity between clock and throughput rates. The clock rate may be fast enough to provide the necessary sampling rate, but it will present problems in system design, both in delivering a very fast clock rate and controlling the power consumption, particularly dynamic power consumption, as this is directly dependent on the clock rate.

Parallelism

The nature of DSP algorithms are such that high level of parallelism are available. For example, the expression in Equation (4.1) can be implemented in a single processor, or a parallel implementation, as shown in Figure 4.1, where each element in the figure becomes a hardware element therefore implying 127 registers for the delay elements, 127 multipliers for computing the products $a_i x_{n-i}$ where $i = 0, 1, 2, \ldots, N-1$ and an 128-input addition which will typically be implemented as an adder tree. In this way, we have the hardware complexity to compute an iteration of the algorithm in one sampling period. Obviously, a system with high levels of parallelism and the needed memory storage capability will accommodate this computation in the time necessary. There are other ways to derive the required levels of parallelism to achieve the performance which is the focus of later chapters.

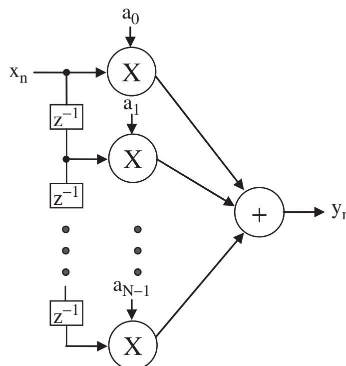


Figure 4.1 Simple parallel implementation of a FIR filter

Data Independency

The data independent property is important as it provides a means for ordering the computation. This can be highly important in reducing the memory and data storage requirements. For example, consider N iterations of the FIR filter computation of Equation (4.1), below:

$$
\begin{aligned}
y_n &= a_0\underline{x_n} + a_1x_{n-1} + +a_2x_{n-2} + \ldots + +a_{N-1}x_{n-N+1} \\
y_{n+1} &= a_0x_{n+1} + a_1\underline{x_n} + +a_2x_{n-1} + \ldots + +a_{N-1}x_{n-N+2} \\
y_{n+2} &= a_0x_{n+2} + a_1x_{n+1} + +a_2\underline{x_n} + \ldots + +a_{N-1}x_{n-N+3} \\
\vdots &= \vdots \\
y_{n+N-1} &= a_0x_{n+N-1} + a_1x_{n+N} + +a_2x_{n+N+1} + \ldots + +a_{N-1}\underline{x_n}.
\end{aligned}
$$

It is clear that the x_n data is required for all N calculations and there is nothing to stop us performing the calculation in such a way that N computations are performed at the same time for $y_n, y_{n+1}, \ldots, y_{n+N-1}$, using the x_n data and thus removing any requirement to store it. Obviously the requirement is now to store the intermediate accumulator terms. This obviously presents the designer with a number of different ways of performing system optimization and in this case, gives in a variation of schedule in the resulting design. This is just one implication of the data independence.

Arithmetic Requirements

In many DSP technologies, the wordlength requirements of the input data are such that the use of internal precision can be considerably reduced. For example, consider the varying wordlengths for the different applications as illustrated in Table 3.6. Typically, the input wordlength will be determined by the precision of the A/D device creating the source material, or the amount of noise in the original source which can have an impact on the total noise in the system. Depending of the amount and type of computation required, e.g. multiplicative or additive, the internal word growth can be limited, which may mean that a suitable fixed-point realization is sufficient.

The low arithmetic requirement is vital, particularly for FPGA implementations where as of yet, no dedicated floating point flexibility is available. This limited wordlength means small memory requirements, faster implementations as adder and multiplier speeds are governed by input wordlengths, and smaller area. For this reason, there has been a lot of work involved in determining maximum wordlengths as discussed in the previous chapter. One of the interesting aspects is that for many processor implementations, both external and internal wordlengths will have been predetermined when developing the architecture, but in FPGAs, it may be required to carry out detailed analysis to determine the wordlength at different parts of the DSP system (Constantinides *et al*. 2004).

All of these characteristics of DSP computation are vital in determining an efficient implementation, and have in some cases, driven technology evolution. For example, one the main differences between the early DSP processors and microprocessor, was the availability of a dedicated multiplier core. This was viable for DSP processors as they were targeted at DSP applications where multiplication is a core operation but this is not the case for general processing applications, and so multipliers were not added to microprocessors at that time.

4.4 Processor Classification

The technology for implementing DSP ranges from microcontrollers, right though to single chip DSP multiprocessors which range from conventional processor architectures with a very long

Table 4.1 Flynn's taxonomy of processors

Class	Description	Examples
Single instruction single data (SISD)	Single instruction stream operating on single data stream	Von Neumann processor
Single instruction multiple data (SIMD)	Several processing elements , operating in lockstep on individual data streams	VLIW processors
Multiple instruction single data (MISD)		Few practical examples
Multiple instruction multiple data (MIMD)	Several processing elements operating independently on their own data streams	Multiprocessor

instruction word (VLIW) extension to allow instruction level parallelism, through to dedicated architecture defined for specific application domains. Although, there have been other more comprehensive classifications after it, Flynn's classification is the most widely known and used, identifying the instruction and the data as two orthogonal streams in a computer. The taxonomy is summarized in Table 4.1.

4.5 Microprocessors

The classical Von Neumann (vN) microprocessor architecture is shown in Figure 4.2. These types of architecture are classical SISD type architectures, sequentially evaluating a list of instructions to apply a variety of instructions to specified data in turn. The architecture consists of five types of unit: a *memory* containing data and instructions, an *instruction fetch and decode* (IFD) unit, the *arithmetic logic* unit (ALU), and the *memory access* (MA) unit. These units correspond to the four different stages of processing, which repeat for every instruction executed on the machine.

1. Instruction fetch
2. Instruction decode
3. Execute
4. Memory access

During the *instruction fetch* (IF) stage, the IFD unit loads the instruction at the address in the program counter (PC) into the instruction register (IR). In the second, *instruction decode* (ID) stage, this instruction is decoded to produce an opcode for the ALU and the addresses of the two data operands, which are loaded into the input registers of the ALU. During the *execute* stage (E), the ALU performs the operation specified by the opcode on the input operands to produce the result, which is written back into memory in the *memory access* (MA) stage. In general, these types of SISD machine can be subdivided into two categories, depending on their instruction set style. *Complex instruction set computer* (CISC) machines have complex instruction formats which can become highly specific for specific operations. This leads to compact code size, but can complicate pipelined execution of these instructions. *Reduced instruction set computer* (RISC) machines, on the other hand, have regular, simple instruction formats which may be processed in a regular manner, promoting high throughput via pipelining, but will have increased code size. The vN processor architecture is designed for general purpose computing, and is limited for embedded applications, due to its highly sequential nature. This makes this kind of processor architecture suitable, for general-purpose environments. However, whilst embedded processors must be flexible, they are

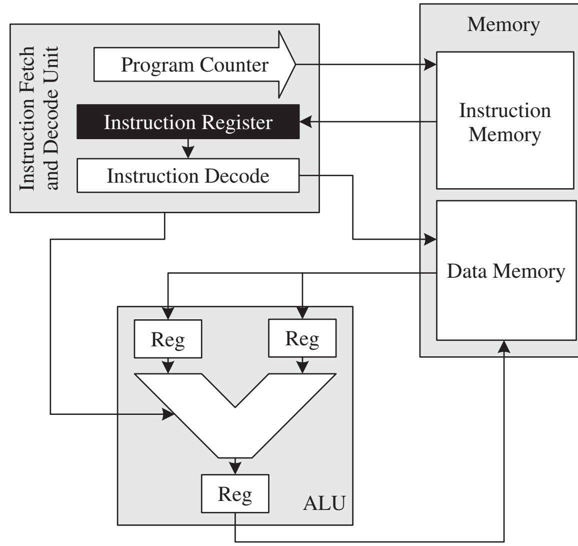


Figure 4.2 Von Neumann processor architecture

often tuned to a particular application and require advanced performance requirements, such as low power consumption or high throughput.

4.5.1 The ARM Microprocessor Architecture Family

The ARM family of embedded microprocessors are a good example of RISC processor architectures, exhibiting one of the key trademarks of RISC processor architectures, namely that of instruction execution path pipelining. Table 4.2 outlines the main characteristics of three members of the ARM processor family.

The increasingly low pipelines in these processor architectures (as identified for the ARM processors in Table 4.2 and Figure 4.3) are capable of enabling increased throughput of the unit, but only up to a point. With increased pipeline depth comes increased control complexity, a limiting factor and one which places a limit on the depth of pipeline which can produce justifiable performance improvements. After this point, processor architectures must exploit other kinds of parallelism, for increased real-time performance. Different techniques and exemplar processor architectures to achieve this are outlined in Section 4.6.

Table 4.2 ARM Microprocessor family overview

Member	Pipeline depth	Description
ARM7	3	Fetch, Decode, Execute
ARM9	5	Fetch, Decode, ALU, Memory Access, Write Back
ARM11	8	See Figure 4.3

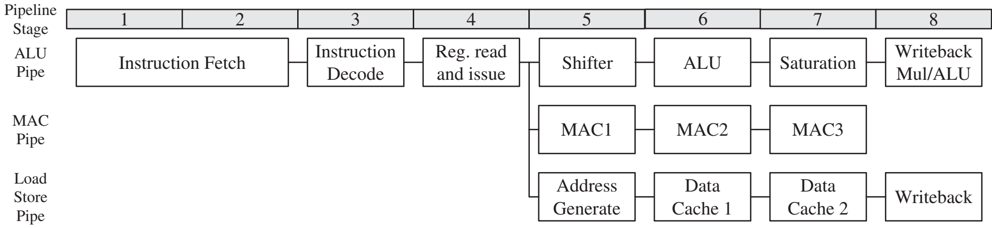


Figure 4.3 ARM11 pipeline architecture

4.6 DSP Microprocessors (DSPμs)

As was demonstrated in the previous section, the sequential nature of the microprocessor architecture makes it unsuitable for the efficient implementation of computationally complex DSP systems, either in that it cannot achieve the required sampling rate, or it meets the requirement, but consumes a lot of power. For microprocessor implementations, the serial architecture is such that for data processing applications, a lot of the transistors will not be performing any useful part in the computation being performed. Thus, the sacrifice of the fixed nature of a general processor means there are a lot of transistors that are consuming power, but which are not contributing to the performance.

This spurred the motivation to look at other types of processor architectures for performing DSP. In the 1980s, DSPμs such as the TMS32010 from Texas Instruments Inc. emerged, which had similar functionality to microprocessors, but differed in that they were based on the Harvard architecture, with separate program and data memories and separate buses. Crudely speaking, they were microprocessor architectures which had been optimized for DSP that perform multiply and accumulation operations, consuming less power. Figure 4.4 shows the difference between the Von Neumann and Harvard architecture. In the Von Neumann machine, one memory is used for both code and data, effectively providing a memory bottleneck. In the Harvard architecture, data memory

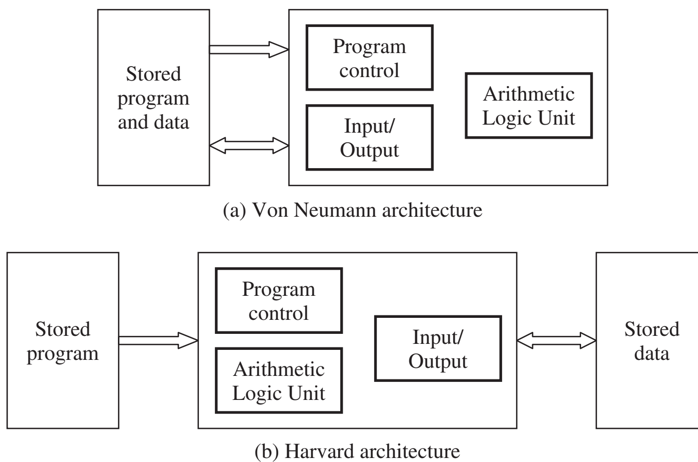


Figure 4.4 Von Neumann and Harvard architectures

and program memory are separate, allowing the program to be loaded into the processor independently from the data. The Harvard architecture has also some dedicated hardware for performing specific operations on the data. Initially, this was a dedicated multiplier, but with increasing level of integration, more complex functions have been added.

Separate data and program memories and dedicated DSP hardware have become the cornerstone of earlier DSP processors. The Texas Instrument's TMS32010 DSP (Figure 4.5) which is recognized as the first DSP architecture, was an early example of the Harvard architecture and highlights these features. It comprises program and data buses which can be clearly seen and even with this early device, a dedicated multiply–accumulate functions was included in addition to the normal arithmetic logic unit (ALU). The earlier TMS32010 16-bit processor had a 200 ns instruction cycle (5 MIPS) and could perform a multiply–accumulate (MAC) operation in 400 ns.

Since this early devices, a number of modifications have occurred to this original architecture (Berkeley Design Technology 2000) which are listed below; the TI TMS320C64xx series device is described on the next section (Dahnoun 2000, Texas Instruments Inc. 1998).

VLIW. Modern processor architectures have witnessed an increase the internal bus wordlengths. This allows a number of operations performed by each instruction in parallel, using multiple processing function units. If successful, the processor will be able to use this feature to exploit these multiple

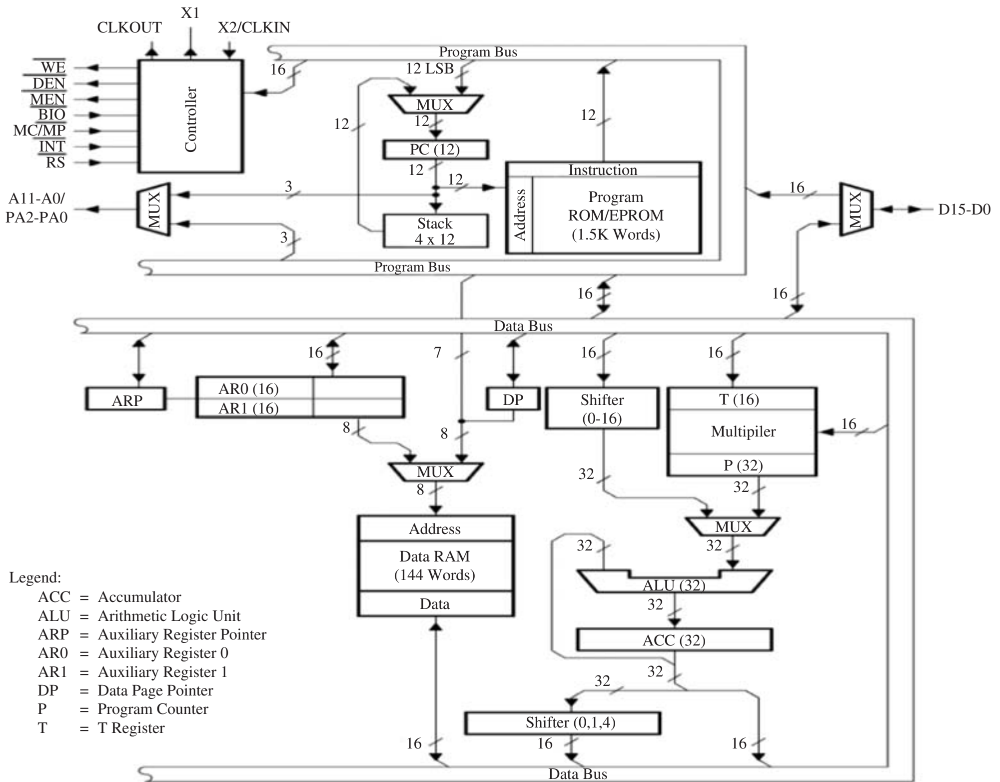


Figure 4.5 Texas Instruments TMS32010 DSP processor. Reproduced by permission of Texas Instruments

hardware units; this depends on the computation to be performed and the efficiency of the compiler in utilizing the underlying architecture. This is complicated by the move toward higher-level programming languages which requires good optimizing compilers that can efficiently translate the high-level code and eliminate any redundancies introduced by the programmer.

Increased number of data buses. In devices such the Analog Devices super harvard architecture (SHARC) ADSP-2106x (Analog Devices Inc. 2000), the number of data buses have been increased. The argument is that many DSP operations involve two operands, thus requiring three pieces of information (including the instruction) to be fed from memory. By increasing the number of buses, a speed-up is achieved, but this also increases the number of pins on the device. However, the SHARC architecture gets around this by using a program cache, thereby allowing the instruction bus to double as a data bus, when the program is being executed out of the program cache.

Pipelining. Whilst the introduction of VLIW has allowed parallelism, another way to exploit concurrency is to introduce pipelining, both within the processing units in the DSP architecture, and in the execution of the program. The impact of pipelining is to break the processing into smaller time units, thereby allowing several overlapping computations to take place at once, in the same hardware. However, this comes at the expense of increased latency. Pipelining can also be employed within the processor control unit (Figure 4.6) which controls the program fetch, instruction dispatch and instruction decode operation and is described in detail, in the next section.

Fixed point operations. Many practical DSP systems only require fixed-point arithmetic and do not require the full precision arithmetic offered by some DSP processing units. For this reason, fixed- and floating-point DSP micros have evolved to match application environments. However, even in fixed-point applications, some operations do not require the full fixed-point range of some processor, e.g. 32 bits in the TMS320C64xx series processor and therefore inefficiency exists. For example, for a filter application in image processing applications, the input wordlength may vary between 8 and 10 bits, and coefficients could take the range 12–16. Thus, the multiplication stage will not require anything larger than a 16×16 multiplier. This is exploited by the DSP processors by organizing the processing unit such as in the TMS320C64xx, by allowing two 16×16 bit multiplications to be take place at the one time, thereby improving throughput. Thus, the processors are not compromised in terms of the internal wordlength used.

These optimizations have evolved over a number of years, and have led to improved performance. However, it is important to consider the operation, in order to understand how the architecture performs in some applications.

4.6.1 DSP Micro-operation

In this section, the TMS320C64xx series architecture (Dahnoun 2000, Texas Instruments Inc. 1998) has been chosen as it is indicative of typical DSP processors. The simplified model of the CPU of the processor is given in Figure 4.6. It comprises the program control unit (PCU) which fetches the instructions, dispatches them to the required processor and then performs the decode for the instruction. In the TMS320C64xx series device, there are two datapath units. In the case of the TI device, the function units are grouped into two sets of four (L, M, S and D) where the L unit can be used for 32/40-bit arithmetic and compare operations, 32-bit logical operations, normalization and bit count operations and saturated arithmetic for 32/40-bit operations, the M unit can perform various types of multiply operation, e.g. quad 8×8, dual 16×16 and single 16×32 operations, the S unit can perform various arithmetic, shift, branch and compare operations and the D unit, various load and store operations. The reader should note that this is not an exact definition of the processing units (Texas Instruments Inc. 1998), but gives some idea of the functionality. The

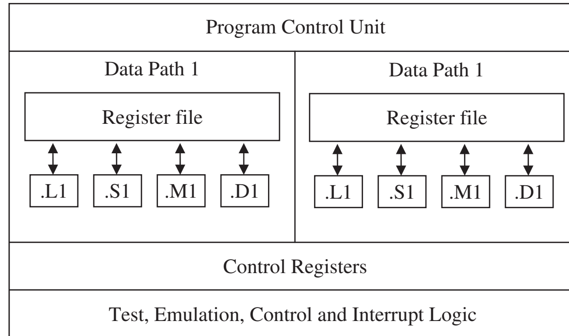


Figure 4.6 Texas Instruments TMS320C62 and TMS320C67 block diagram(Dahnoun 2000)

processor also contains a register file and multiple paths for communication between each block. In the case of the TI processor, there are cross-paths to allow linking of one side of the CPU to the other (Dahnoun 2000, Texas Instruments Inc. 1998).

The key objective is to be able to exploit the processing capability offered by this multiple hardware which depends both on the computation to be performed and the use of optimizing compilers that perform a number of simplifications to improve efficiency. These simplifications include routines to remove all functions that are never called, to simplify functions that return values that are never used, to reorder function declarations and propagate arguments into function bodies (Dahnoun 2000). The compiler also performs a number of optimizations to take advantage of the underlying architecture including software pipelining, loop optimizations, loop unrolling and other routines to remove global assignments and expressions (Dahnoun 2000).

4.7 Parallel Machines

Whilst the sequential model has served well in the sense that it can implement a wide range of algorithms, the real gain from DSP implementation comes from parallelism of the hardware. For this reason there has been a considerable interest in developing hardware involving parallel hardware evolving from the early days of the transputer (Inmos 1989). However, it is capturing this level of parallelism that is the key issue. A key architecture which was developed to capture parallelism is the systolic array (Kung and Leiserson 1979, Kung 1988) which forms the starting point for this section.

4.7.1 Systolic Arrays

Systolic array architectures were introduced into VLSI design by Kung and Leiserson in 1978 (Kung and Leiserson 1979). In summary, they have the following general features (Kung 1988):

- an array of processors with extensive concurrency
- small number of processor types
- control is simple
- interconnections are local

Their processing power comes from the concurrent use of many simple cells, rather than the sequential use of a few very powerful cells. They are particularly suitable for parallel algorithms with simple and regular dataflows, such as matrix-based operations. By employing pipelining, the

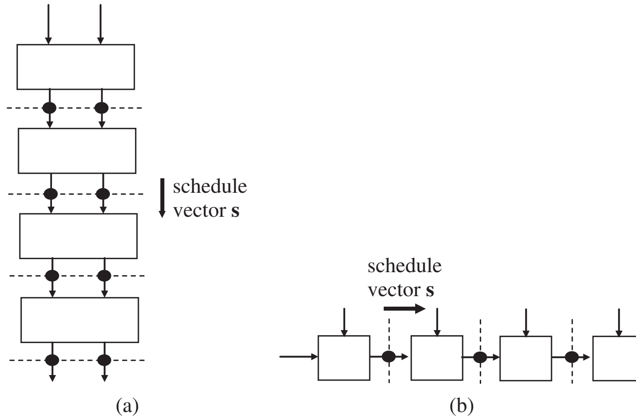


Figure 4.7 Linear systolic array architectures: (a) column; (b) row

operations in the systolic array can be continually filtered through the array, enabling full efficiency of the processing cells.

The examples in Figure 4.7 show a simple case of a systolic array with a linear structure. Here, the black circles represent pipeline stages after each processing element (PE). The lines drawn through these pipeline stages are the scheduling lines depicting which PEs are operating on the same iteration at the same time; in other words, these calculations are being performed at the same clock cycle.

Figure 4.8 shows two more examples. Figure 4.8(a) is a rectangular array of cells, each with local interconnects. This type of array is highly suitable to matrix operations. Figure 4.8(b) gives an example systolic array with a hexagonal structure. In all cases illustrated, each PE receives data only from its nearest neighbour and each processor contains a small element of local memory on which intermediate values are stored. The control of the data through the array is by a synchronous clock, effectively pumping the data through the array hence giving them the name of 'systolic' arrays due to the analogy of a heart pumping blood around the body. Figure 4.9 depicts the systolic

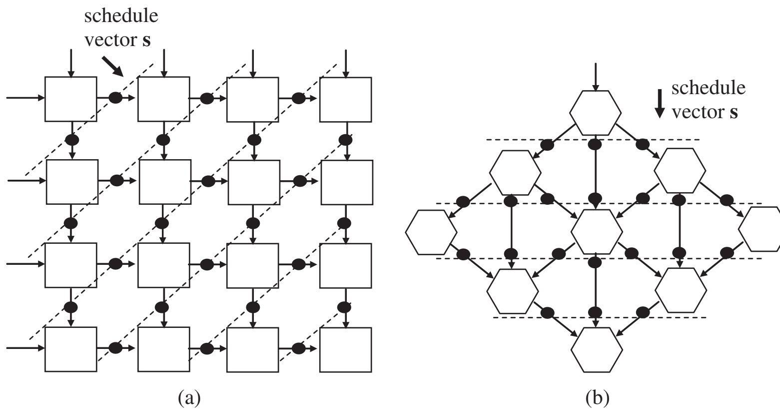


Figure 4.8 Systolic array architectures: (a) rectangular (b) hexagonal

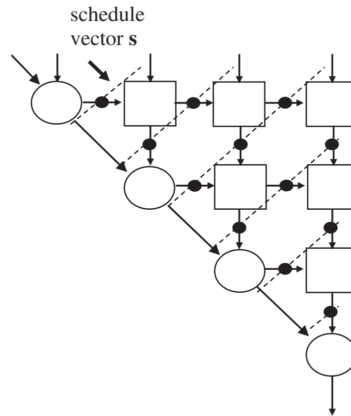


Figure 4.9 Triangular QR systolic array

array applied for QR decomposition. The array is built from two types of cells, boundary and internal, all locally interconnected.

The concept of systolic arrays was employed in many ways. McCanny and McWhirter (1987) applied it at the bit level whereas the original proposer of the technique, developed the concept into the *iWarp* which was an attempt in 1988 by Intel and Carnegie Mellon University to build an entirely parallel computing *node* in a single microprocessor, complete with memory and communications links. The main issue with this type of development was that it was very application specific, coping with a range of computational complex algorithms; instead, the systolic array design concept was applied more successfully to develop a wide range of signal processing chips (Woods and Masud 1998, Woods *et al*. 2008) and indeed, Chapter 12 demonstrates how the concept has been successfully applied to the development of an IP core.

4.7.2 SIMD Architectures

The first kinds of parallel computers based on the SIMD architectures included the Illiac IV (Barnes *et al*. 1968) and the connection machine-2 or CM-2 (Hillis 1985). The Illiac IV stood for the Illinois integrator and automatic computer and was a highly parallel machine with 64 processing engines (PEs), all controlled by one Control Unit (CU). The PEs implemented the same operation simultaneous, and each PE had its localized memory. The original CM-2 concept came from MIT, and involved a hypercube of simple processing engines comprised of simple CPUs with their own memory. The CM-2 had quite of number of simple PEs (typically 64 000) which processed one bit at a time, but was later extended to floating point. These early architectures fell into the category of the general SIMD-type architectures, in that they consisted of one CU, fetching and issuing instructions to a number of processing elements (PEs) which performed the same operations simultaneously (Sima *et al*. 1997). The major properties that differentiate many architectures classified as SIMD types are PE complexity, degree of PE autonomy, PE connection types and number and connection topology of the PEs (Sima *et al*. 1997). Essentially, the SIMD model applies very well to regular structure of computations which are encountered in a range of signal and image processing applications. However, the development of traditionally massively parallel SIMD type machines required custom implementation which were specialized for certain algorithms or applications. This was expensive and compared poorly against the development of the complex microprocessor which was now being driven by evolving silicon technology. For this reason, traditional SIMD type

architectures lost ground to the MIMD-type architectures which could be built from cheap, off-the-shelf microprocessors.

More recently, SIMD type architectures have become relevant again, although in a much smaller scale as instruction set extensions to handle multimedia workloads in modern microprocessors. Examples of such instruction set extensions include Streaming SIMD Extensions (SSE2) provided by Intel initially for their Pentium 3 microprocessor families (Intel Corp. 2001) and Apple, IBM, and Motorola's combined offering for Altivec (Intel Corp. 2001), allowing 128 bits to be processed at a time translating to 16 (8-bit) or 8 (16-bit) integer operations at a time, or 4 floating operations. These new instruction sets imply architectural changes to be made to the underlying processor in terms of processing and register file organization, and also increases in the instruction sets. Additionally, new single-chip dedicated SIMD architectures have appeared such as Imagine processor (Khailany *et al*. 2001) and Clearspeed's CSX600 (Clearspeed Tech. plc 2006). The efficient and high-level programming of these platforms is still a hot research issue, but the two processors are described in a little more detail.

Imagine Processor

The Imagine chip is a parallel architecture which comprises 48 floating-point ALUs and a special memory hierarchy, optimized for stream-based programs (Kapasi *et al*. 2002). The Imagine project effort comprises, not only the architecture development, but a programming environment based on a 'streaming programming' model. The key in developing both the programming environment *and* the platform is vital in achieving an efficient implementation platform. The streaming programming model is ideal for image processing applications which exhibit high levels of data streaming, due to the need to pass around image data which is typically large. This model allows the software to exploit the locality and parallelism inherent in many image processing applications, allowing high performance to be gained from the underlying architecture. An expected processing performance of 18.3 GOPS is quoted for MPEG-2 encoding applications, corresponding to 105 frames per second on a 720×480 pixel, 24-bit colour image whilst dissipating 2.2 W (Khailany *et al*. 2001).

A block diagram of the Imagine Stream Processor architecture is shown in Figure 4.10. It comprises a 128-kbyte stream register file known as a SRF, 48 floating-point arithmetic units organized into 8 arithmetic clusters which are controlled by a microcontroller, a network interface, a streaming memory system with 4 SDRAM channels, and a stream controller. A series of stream instructions have been created: *load* and *store* which load (and store) data from (and to) the off-chip SDRAM to (and from) the SRF, allowing access to external data; *send* and *receive* instructions which send and receive streams of data from the SRF to other Imagine processors or processing elements connected via the external network; a *Cluster op* instruction which loads the streams to the processing units and then stores the streams back to the SRFs and *Load microcode* instruction which loads streams consisting of kernel microcode, 576-bit VLIW instructions from the SRF into the microcontroller instruction store (a total of 2,048 instructions, (Khailany *et al*. 2001). Imagine supports streams of 32 k words long. The processor is set up by the host processor which sends stream instructions to the stream controller, which are then decoded and commands then issued to other on-chip modules.

One of the key attractions of Imagine is that the processing kernels can perform compound operations; these read an element from the input stream in the SRF and then perform multiple arithmetic operations before appending the results to output streams and transferring them back to the SRF. This avoids the various series of *fetch* and *execute* instructions of sequential processor and considerably reduces the data transfers as the data can be used multiple times in one processing stage. This aspect was similar to the concepts of systolic arrays (Kung 1988) which reduce bandwidth communications by reusing the data when available in the processor.

A number of examples have been presented which demonstrate the capability of the processor. In the case of 7×7 *convolution*, Khailany *et al*. (2001) show how Imagine can load data from

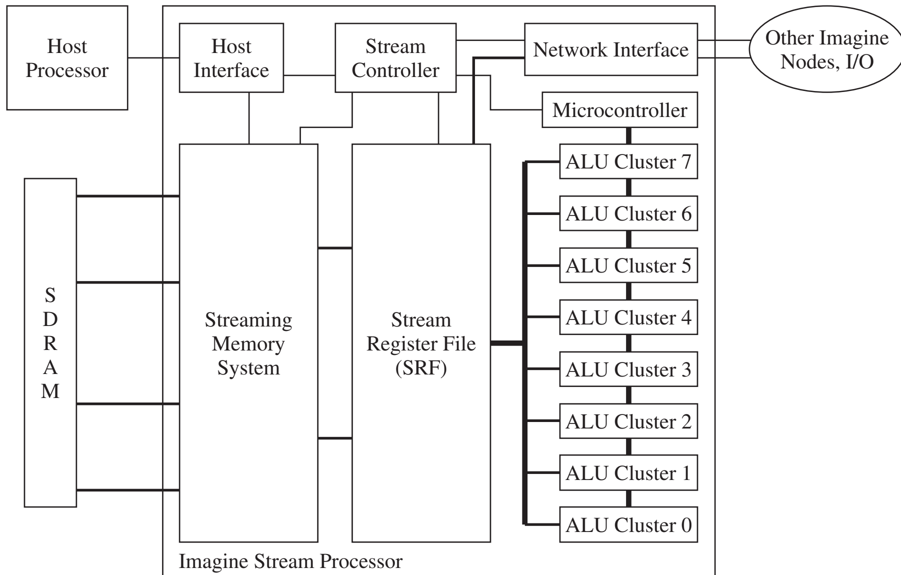


Figure 4.10 Imagine stream processor architecture (Khailany *et al.* 2001)

the external RAM as a row of pixels and then distribute these pixels individually to the each of the eight processors, along with the previously stored data, namely an accumulating partial product which is loaded from the SRFs. Each of these operations is then computed until the full operation is complete. This demonstrates a number of highly important features of Imagine which is central to achieving a high performance, namely loading of data efficiently, reuse of data within the processing engine and efficient exploitation of parallelism within the processing engine.

The choice of 8 processors is related to the fact that the core image block size of many applications is a 8 × 8 image block and provides an efficient match between the processing requirements and the computation needs. Thus for smaller image processing operations such as the 3 × 3 block used in various image processing operations such as Sobel and Laplace filters, there will be an under-utilization of the performance.

A hierarchical approach to bandwidth optimization has been used with the highest bandwidth capability reserved for the processing engine, namely 544 Gbytes/s which is then reduced to 32 Gbytes/s (SRF bandwidth) and even further to 2.67 Gbytes/s for the off-chip bandwidth to the SDRAM. Many of the optimizations employed to achieve performance gain in FPGA implementations, which are developing by creating the detailed circuit architecture, have been employed in the Imagine processor, but admittedly for a limited range of application. This highlights the on-going trend, as will be seen in the latest family of FPGA devices, to develop devices that are specific to a range of applications and ties in with the discussion in the first chapter.

Storm Stream Processor

The concept of the Stream processor has been captured in the Storm Processor by Stream Processors Inc., (SPI), which is a fabless semiconductor company that has spun off from the Stanford research. The *Storm-1* processor family is argued to deliver considerable improvement over other DSP technologies as demonstrated by Table 4.3.

Table 4.3 Storm comparison with other processors(Stream Processors Inc. 2007)

Supplier device	SPI Storm-1 SP16HP	Xilinx Virtex™-5 5VSX35T	Altera Stratix III EP3SL70	TI TMS320 C6454
Architecture	DSP	FPGA	FPGA	DSP
Clock Frequency	700 MHz	550 MHz	300 MHz	1 GHz
GMACS/chip	112	106	86	8
Design methodology	Software programmable	Reconfigurable hardware	Reconfigurable hardware	Software programmable

The Storm-1 processor comprises a host CPU (System MIPS) for system-level tasks and a DSP Co-processor Subsystem with a DSP MIPS which runs the main threads that make kernel function calls to the *data parallel unit* (DPU). The DPU comprises a data parallel unit with 16 or 8 data-parallel *execution lanes* (depending on the device chosen) compared with 8 in the Imagine processor. There are 5 ALUs in each *lane* or cluster and an instruction fetch unit and VLIW sequencer which would seem to capture aspects of the Stream register file. The DPU Dispatcher receives kernel function calls to manage runtime kernel and stream loads. As in the Imagine processor, one kernel at a time is executed across the lanes, operating on local stream data stored in the *lane register files*. Each lane has a set of VLIW ALUs and distributed operand register files (ORF) allow for a large working data set and high local bandwidth. The *interlane switch* is a full crossbar for high-speed access between lanes which is scheduled at compile time and would seem to represent an enhancement on the basic bus interconnection from the SRF in the Imagine processor. At the time or writing, the device is priced at US$149 for a volume of 10 000 units.

As can be seen from the performance figures, the Storm-1 processor considerably outperforms the TI processor, given that it has been developed with clear exploitation of parallelism in mind from the start and the notion of a simple programming models to harness this power. However, the FPGAs have not performed that badly, given the lower processing speed. The table overall reiterates the importance of being able to harness the circuit architecture development in a way to capture the performance.

Clearspeed CSX600 Processor

The CSX600 (Clearspeed Tech. plc 2006) is an embedded, low-power, data-parallel co-processor from Clearspeed. The technology is targeted at fine-grained operations such as matrix and vector operations, unrolled independent loops and multiple simultaneous data channels; these are operations where parallel processing will achieve gains and which represent classical SIMD computations. It would appear that the architecture is now targeted at high-performance computing applications, providing an acceleration for BLAS (basic linear algebra subprograms) libraries, standardized application programming interfaces for subroutines to perform linear algebra operations, e.g. vector and matrix multiplication, and the Linear Algebra PACKage (LAPACK) which is a software library for numerical computing. Several examples are quoted for accelerating Matlab simulations and force field in molecular dynamics (Amber 9 Sander Implicit methods). It is programmed in C, but

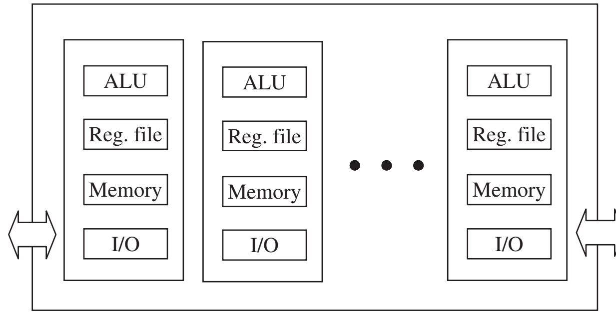


Figure 4.11 Clearspeed CSX 'poly' execution unit

various libraries are provided to allow it to allow C++ and Fortran applications, to interact with the processor.

The CSX600 provides 25 GFLOPS of sustained single or double precision floating-point performance, while dissipating an average of 10 W. The architecture uses 64-bit addressing, thus supporting access to multi-gigabyte DDR2 SDRAMs via a local ECC protected memory interface, to provide the necessary memory bandwidth. Central to the CSX600 architecture is a processing engine called a multi-threaded array processor (MTAP), which allows parallel computation to take place on a core DSP function, and the ClearConnectTM Network on Chip (NoC) technology addressing the needs of a processing engine, in getting data to and from, the processing engine via the high bandwidth.

The architecture comprises an MTAP processor core comprising 96 high-performance 'poly' PE cores, each with 6 kbytes of dedicated memory, allowing local memory for each of the PEs; 128 kbytes on-chip scratch pad memory; an external 64-bit DDR2 DRAM interface; 64-bit virtual, 48-bit physical addressing and various instruction and data caches all of which are interconnected via the ClearConnect on-chip network. The poly PE structure is given in Figure 4.11. One part of the PE performs one-off operations and handles program control such as branch and thread switching whereas the computational aspects that handle the highly parallel computation, are carried out in the rest of the core.

4.7.3 MIMD Architectures

MIMD type architectures consist of a collection of processors that are interconnected by various network topologies, each having its own control units that issue instructions. The major categorization among different MIMD architectures is based on memory organization. A shared memory model is one in which a number of processors equally share a memory space and communicate by reading and writing locations in the shared memory. The memory would then have to be equally accessible by all processors. A distributed memory model is one, in which processors have their own local memories and message-passing is used for the communication among the processors (Sima *et al.* 1997). This avoids any memory contention as each distributed processor will have main access to the memory, and they are more scalable, as memory access bandwidth is limited by the interconnection network in shared-memory architectures. Because of the scalability restrictions, shared memory MIMD architectures have fewer processors than those of the distributed memory MIMD architectures which can scale up to massively parallel machines.

Computer clusters, where a farm of computers (PCs, workstations or symmetric multiprocessors (SMPs)) are connected to each other over networks implemented with high-speed interconnect

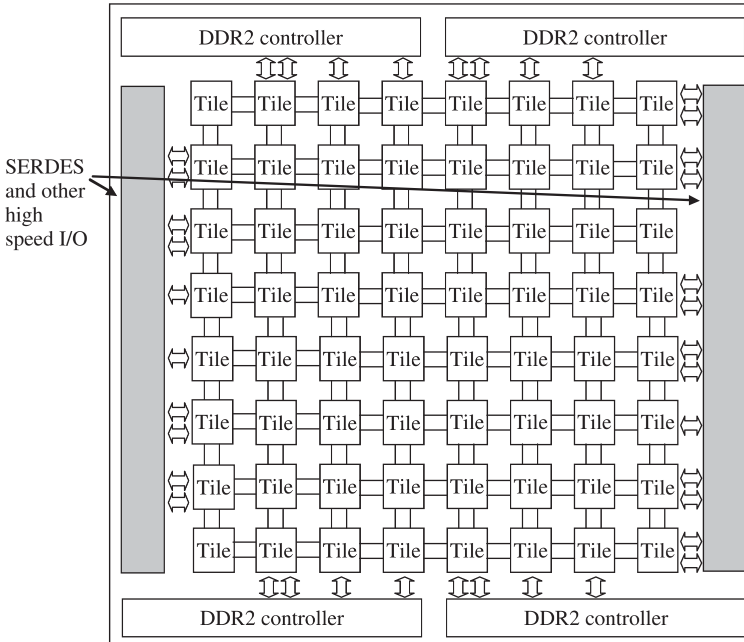


Figure 4.12 Tilera TILE architecture (Tilera Corp. 2007)

technologies such as Myrinet (Boden *et al.* 1995) are considered as distributed memory MIMD architectures. These architectures provide a lower-cost solution compared with the massively parallel computers, albeit with a much slower interconnect.

TILE64™ Multi-core System

A recent example of a multi-core processor is the TILE64™ which comprises 64 identical processor cores called *tiles*, interconnected using a propriety on-chip network topology called Mesh™. The TILE64™ architecture is shown in Figure 4.12. Each tile is a complete full-featured processor which includes integrated level 1 (L1) and level 2 (L2) cache, and a nonblocking switch that connects the tile into the mesh. The localization of the memory, means that each tile can run its own OS separately, or multiple tiles taken together, can run a multi-processing operating system like SMP Linux (Tilera Corp. 2007).

4.8 Dedicated ASIC and FPGA Solutions

Up to now, the DSP technology offerings have been in the form of some type of pre-defined architectural offering. The major attraction of dedicated ASIC offerings (which largely apply to FPGA realizations) is that the architecture can be developed to specifically match the algorithmic requirements, allowing the level of parallelism to be created to ultimately match the performance requirements. Take for example, the 128-tap FIR filter example quoted earlier. With FPGA and ASIC implementations, it is possible to dedicate a multiplier and adder to each multiplication and

addition respectively, thereby creating a fully parallel implementation. Moreover, this can then be pipelined in order to speed up the computation and indeed, duplicated N times to give a N-fold speed improvement. The concept of how the designer can create the hardware necessary to give the required performance is covered in detail in Chapter 8.

When considering the programmability argument, these types of ASIC solutions will not have been developed to include additional hardware to provide programmability as performance (speed, area and power) has probably been the dominant aspect, to justify using the technology in the first place. Moreover, additional levels of programmability cause an increase in test and verification times. Non-recurrent engineering (NRE) costs are such that the cost to produce a number of prototypes is now typically in excess of 1B$. Thus the argument for using dedicated ASIC hardware has therefore got to be compelling.

However, by their very nature, FPGA solutions avoid these high NRE costs by giving the user a part that can be programmed. Whilst the area, speed and particularly power performance is not as compelling, the notion that the part can be programmed or configured, avoids the NRE cost problems, but also considerably the design risk as the part can be reconfigured if mistakes have been made in the design process.

Crudely speaking, FPGAs can be viewed as comprising:

- programmable logic units that can be programmed to realize different digital functions
- programmable interconnect to allow different blocks to be connected together
- programmable I/O pins.

This presents a huge level of programmability allowing functions that are implemented on the FPGA to be changed, or existing functions on the FPGA to be interconnected in a different way or indeed, circuits on different FPGAs to be interconnected in different ways. In the earlier days, the ability to change the interconnection was the main attraction, but as FPGAs have grown in complexity, this has been supplemented by changing actual functionality. Given that the aim is to create highly efficient designs, the main aim is to utilize the huge processing resource in the most efficient manner, allowing the solution to outperform other technology offerings. Like ASIC, this requires the design of a suitable circuit architecture to best utilize this underlying hardware. Historically, this has been viewed as a hardware design process which compared with software, is long and involved. However, the performance advantages that some of the examples given later in the book will demonstrate are quite large and need to be taken into consideration, which is the main focus of this book.

4.9 Conclusions

The chapter has highlighted the variety of different technologies used for implementing DSP complex systems. These compare in terms of speed and power consumption and, of course area, although this is a little difficult to ascertain for processor implementations. The chapter has taken a specific slant on programmability with regard to these technologies and in particular, has highlighted how the underlying chip architecture can limit the performance. Indeed, the fact that it is possible to develop application-specific circuit architectures for ASIC and FPGA technologies is the key feature in achieving the high performance levels. It could be argued that the fact that FPGAs allow circuit architectures and are programmable are the dual factors that makes them so attractive for some system implementation problems.

Whilst the flavour of the chapter has been to present different technologies and in some cases, compare and contrast them, the reality is that modern DSP systems are now collections

of these different platforms. Many companies are now offering complex DSP boards comprising microprocessors, DSPμs and FPGAs, and companies such as IBM are offering embedded FPGA devices. Given the suitability of different platforms to different computational requirements, this comes as no surprise. Current DSPμs and, as the next chapter will demonstrate, recent FPGA offerings, can be viewed as heterogeneous platforms comprising multiple hardware components.

References

Analog Devices Inc. (2000) Adsp-2106x sharcfi dsp microcomputer family: Adsp-21060/adsp-21060l. Web publication downloadable from http://www.analog.com/.

Barnes GH, Brown RM, Kato M, Kuck DJ, Slotnick DL and Stokes RA (1968) The ILLIAC IV computer. *IEEE Trans. Computers* **C-17**, 746–757.

Berkeley Design Technology (2000) Choosing a DSP processor. Web publication downloadable from http://www.bdti.com.

Boden NJ, Cohen D, Felderman RE, Kulawik A, Seitz C, Seizovic J and Su WK (1995) Myrinet: A gigabit-per-second local area network. *IEEE Micro* pp. 29–36.

Clearspeed Tech. plc (2006) Csx600 datasheet. Web publication downloadable from http://www.clearspeed.com.

Constantinides G, Cheung PYK and Luk W 2004 *Synthesis and Optimization of DSP Algorithms*. Kluwer, Dordrecht.

Dahnoun N (2000) *Digital Signal Processing implementation using the TMS320C6000^{TM} DSP Platform*. Pearson Education, New York.

Hennessey J and Patterson D (1996) *Computer Architecture: a quantitative approach*. Morgan Kaufmann, New York.

Hillis W (1985) *The Connection Machine*. MIT Press, Cambridge, MA, USA.

Inmos (1989) *The Transputer Databook (2nd edn)*. INMOS Limited. INMOS document number: 72 TRN 203 01.

Intel Corp. (2001) Ia32 intel architecture software developers manual volume 1: Basic architecture.

Jackson LB (1970) Roundoff noise analysis for fixed-point digital filters realized in cascade or parallel form. *IEEE Trans. Audio Electroacoustics* **AU-18**, 107–122.

Kapasi U, Dally W, Rixner S, J.D. O and Khailany B (2002) Virtex5.pdf *Proc. 2002 IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors*, pp. 282–288.

Khailany B, Dally WJ, Kapasi UJ, Mattson P, Namkoong J, Owens JD, Chang A and Rixner S (2001) Imagine: Media processing with streams. *IEEE Micro* **21**, 35–46.

Kung HT and Leiserson CE (1979) Systolic arrays (for VLSI) *Sparse Matrix proc. 1978*, pp. 256–282.

Kung SY (1988) *VLSI Array Processors*. Prentice Hall, Englewood Cliffs, NJ.

McCanny JV and McWhirter JG (1987) Some systolic array developments in the UK. *IEEE Computer Special issue on Systolic Arrays*, pp. 51–63.

Sima D, Fountain T and Kacsuk P (1997) *Advanced Computer Architectures:A Design Space Approach*. Addison-Wesley, Harlow, UK.

Stream Processors Inc. (2007) Stream processing: Enabling a new class of easy to use, high-performance DSPs. Web publication downloadable from http://www.streamprocessors.com.

Texas Instruments Inc. (1998) Tms320 DSP development support reference guide. Web publication downloadable from http://www.ti.com/.

Tilera Corp. (2007) Tile64 product brief. Web publication downloadable from www.tilera.com.

Woods R and Masud S (1998) Chip design for high performance DSP. *IEE Elect. and Comms. Jour.* **10**, 191–200.

Woods RF, McCanny JV and McWhirter JG (2008) From bit level systolic arrays to HDTV processor chips. *Journal of VLSI Signal Proc.* Special issue on 20 years of ASAP. DOJ 10.1007/S11265-007-0132-z. ISSN 0 922-5773.

5

Current FPGA Technologies

5.1 Introduction

By describing the details of the various technologies in the previous chapter, it becomes clear that the choice of the specific technology, and the resulting design approach implied by the technology, indicates the level of performance that will be able to be achieved for the specific DSP system under consideration. For example, the use of simple DSP microcontrollers implies a DSP system with relatively low performance requirements, and indicates that the user needs to produce C or C++ code as a source for its implementation. However, it is possible that the user can employ Matlab® or Labview as not only the initial design environment to scope the requirements such as wordlength or number representation for the system, but also to use the design approach and its available software routines, to produce the actual DSP source code for the microcontroller. Whilst it could be argued that the quality of the code produced by such approaches can be inefficient, it is quite possible that it is sufficient to meet the performance requirements of the applications, whilst still using a practical, i.e. low-cost, microcontroller, thus, meeting cost requirements as well, by reducing design time.

This design approach can be applied for the full range of 'processor' style platforms, but it may be required that dedicated *handcrafted* C code is produced, to achieve the necessary performance. This is probably particularly relevant in applications where performance requirements are tight (and cannot be met by the computational complexity of the platform), or the specific structure possesses dedicated functionality not well supported within the high-level tool environment; this is typically the case for the newer reconfigurable or dedicated processors, such as the *Storm* Stream Processor. In these cases, it is clear that the platform has been chosen as it offers some superior performance in terms of an area–speed–power metric; the attraction of the platforms could be in the form of specific features such as multiple MAC units of some commercial DSP platforms or, the dedicated processing functionality of the specialized DSP platform such as the data parallel unit of the *Storm* Stream Processor. In these cases, the user is having to compromise the ease of design, in order to avail of the specific architectural feature offered by the technology.

This notion is taken to extreme in the SoC concept where the user is now faced with creating the circuit architecture to best match the performance requirements of the specific system. The user can now create the system requirements to *ultimately* match the DSP systems requirements. However, this ideal notion is tampered with the practical limitations of being able to create this *ultimate* architecture, and hence, design approaches which either involve a specific, existing architectural style, or which utilize a range of existing building blocks, tend to dominate. This suggests an SoC

FPGA-based Implementation of Signal Processing Systems R. Woods, J. McAllister, G. Lightbody and Y. Yi
© 2008 John Wiley & Sons, Ltd

implementation style, where a limited range of functionality will be employed to create systems within a reasonable time.

As Section 4.8 indicated, this is effectively what an FPGA platform offers; FPGAs have emerged from being a 'glue logic' platform, to become a collection of system components with which the user can create a DSP system. The purpose of this chapter is to give a reasonably detailed description of the current FPGA technologies, with a focus on how they can be used in creating DSP systems. The complete detail is given in the various data sheets available from the variety of companies who sell the technology, such as Xilinx, Altera, Atmel, Lattice and Atmel; however the chapter acts to stress the important features and highlights aspects that are important for DSP implementation. Whilst quite a number of different technologies are available from each company, the focus has been to concentrate of the latest commercial offering such as the Stratix®III family from Altera and the Virtex™-5 FPGA family from Xilinx. In addition, technology that is particularly different from other offerings is also described, such as the ProASIC^{PLUS} FPGA technology from Actel which is based on flash technology and the ispXPLD™ 5000MX family from Lattice which extends the CPLD concept (as compared to the LUT-based approach).

The chapter starts with a brief historical perspective of FPGAs in Section 5.2, describing how they have emerged from being a fine-grained technology to a technology with complex system blocks. Section 5.3 describes the Altera FPGA family, concentrating specifically on their Stratix® III FPGA family, as it represents the most powerful FPGA family that the company offers. The MAX®7000 FPGA technology is also briefly described as it represents the evolution of the PLD concept which was architecture on which Altera based their initial programmable hardware offerings. Section 5.4 then goes on to describe the FPGA technology offerings from Xilinx, specifically the Virtex™ FPGA family. Once again, we concentrate on the most recent FPGA version of this technology, namely the Virtex™-5 FPGA family. With both Altera and Xilinx, we have tried to concentrate on the aspects of the hardware that are very relevant to DSP systems, but also have made an attempt to highlight other aspects such as high speed I/O, clocking strategy and memory organization, as these are also very important in determining overall system performance.

There are a number of other technologies offered by Actel, Atmel and Lattice that offer specific features and are very relevant in certain markets. For example, Lattice offer the ispXPLD™ 5000MX family which offers a combination of E^2PROM nonvolatile cells to store the device configuration and SRAM technology to provide the logic implementation; these are described in Section 5.5. Actel® offer a number of FPGA technologies based on flash and antifuse technologies. Initially, Actel were known for, and still offer, an antifuse technology, namely the Antifuse SX FPGA technology which is described in Section 5.6.1. In Section 5.6.2, the ProASIC^{PLUS} flash, FPGA technology is described. This approach allows the device to store its program and remove the need for a programming device as in SRAM-based technology. The company's most recent offering, the Fusion™ technology which represents the first *mixed signal* FPGA, is also covered. Finally, the Atmel® AT40K FPGA technology is described in Section 5.7, as though a little older, it offers a partially reconfigurable solution. In Section 5.8, some conclusions are given.

5.2 Toward FPGAs

In the 1970s, logic systems were created by building PCB boards comprising of TTL logic chips. However, one the limitations was that as the functions got larger, the size of the logic increased, but more importantly, the number of logic levels increased, thereby compromising the speed of the design. Typically, designers used logic minimization techniques such as those based on Karnaugh maps or Quine–McCluskey minimization, to create a *sum of products* expression which could be created by generating the product terms using AND gates and summing them using an OR gate.

The concept of creating a structure to achieve implementation of this functionality, was captured in the *programmable array logic* (PAL) device, introduced by Monolithic Memories in 1978. The PAL comprised a programmable *AND* matrix connected to a fixed *OR* matrix which allowed sum of products structures to be implemented directly from the minimized expression. The concept of an AND and OR matrix became the key feature of a class of devices known as the *programmable logic devices* family; a brief classification is given in Table 5.1. As illustrated in Figure 5.1, a read only memory (ROM) possesses the same structure, only with a *fixed* AND plane (effectively a decode) and a *programmable* OR plane. In one sense, the structure can be viewed as providing the capability of storing four (in general 2^n) of two- (or m-) bit words, as shown in Figure 5.2. The decoder, which is only required to reduce the number of pins coming into the memory, is used to decode the address input pins and a storage area or memory array is used to store the data. As the decoder generates the various address lines using AND gates and the outputs are summed using OR gates, this provides the AND–OR configuration needed for Boolean implementation. In general, a 2^n by 1-bit ROM could implement any n-input Boolean function; a 4-input ROM or LUT thus became the core component of the very first FPGA, namely the Xilinx XC2000 FPGA. The 4-input LUT was small enough to achieve efficient utilization of the chip area, but large enough to implement a reasonable range of functions. If a greater number of inputs was required, this could be achieved by cascading or parallelizing the LUT inputs. This would result in a slower implementation, but it would provide a better utilization than with larger LUTs.

Table 5.1 PLD types

	AND matrix	OR matrix
ROM	Fixed	Prog
PLA	Prog	Prog
PAL	Prog	Fixed

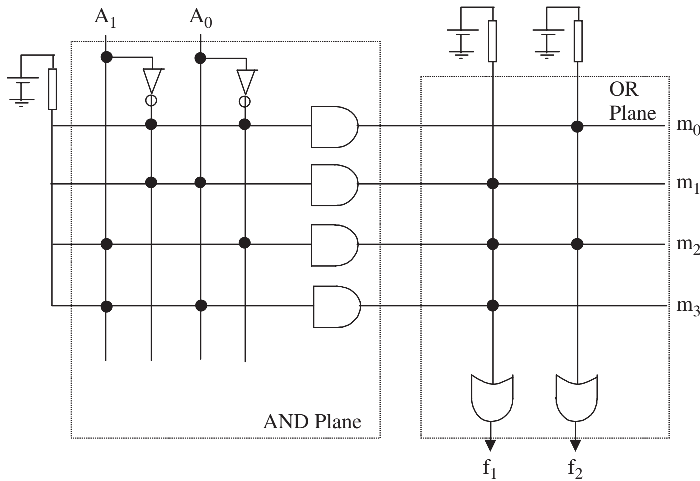


Figure 5.1 ROM detailed structure

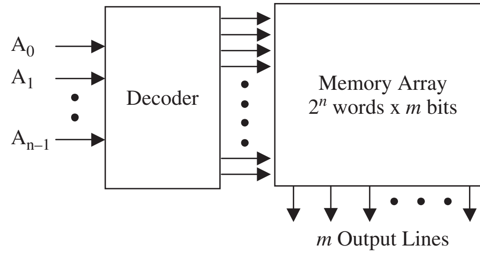


Figure 5.2 ROM block diagram

The PLD structure had a number of advantages. It clearly matched the process of how the sum of products sum was created by the logic minimization techniques. The function could then be fitted into one PLD device, or if enough product terms were not available, could be fed back into a second PLD stage. Another major advantage was that the circuit delay is deterministic either comprising one level of logic level or two, etc. However, the real advantage came in the form of the programmability which reduced the risk in hardware PCB development, allowing possible errors to be fixed by adjusting the logic implementation of the PLD. However, as integration levels grew, the concept of using the PLD as a building block became an attractive FPGA proposition as illustrated by the early Altera offerings and indeed, by their current MAX 7000 device family. As mentioned earlier, Xilinx opted for the ROM or look up table (LUT) approach.

5.2.1 Early FPGA Architectures

The early FPGA offerings comprised a Manhattan style architecture where each individual cell comprised simple logic structures and cells were linked by programmable connections. Thus the FPGA could be viewed as comprising the following:

- programmable logic units that can be programmed to realize different digital functions
- programmable interconnect to allow different blocks to be connected together
- programmable I/O pins.

This was ideal for situations where FPGAs were viewed a glue logic as programmability was then the key to providing redundancy and protection against PCB board manufacture errors, and FPGA components could be used to provide programmable system interconnectivity; this might even provide a mechanism to correct faults caused by incorrect system design. However, technology evolution outlined by Moore's law, now provided scalability for the FPGA vendor. During the 1980s, this was exploited by FPGA vendors in scaling their technology in terms of number of programming blocks, numbers of levels of interconnectivity and number of I/Os. However, it was recognized that this approach had limited scope, as scaling meant that interconnect was becoming a major issue and technology evolution now raised the interesting possibility that dedicated hardware cells could be included, such as dedicated multipliers and more recently, processors. In addition, the system interconnectivity issue would also be alleviated by including dedicated interconnectivity in the form of SERDES and Rapid I/O.

Technology evolution has had a number of implications for FPGA technology:

Technology debate The considerable debate of which technology was effectively determined by Moore's law. In the early days, three different technologies emerged, namely conventional

SRAM, antifuse and EPROM or E^2PROM technologies. In the latter two cases, both technologies required special steps, either to create the antifuse links or to create the special transistors to provide the EPROM or E^2PROM transistor. Technological advances favoured SRAM technology as it only required standard technology; this became particularly important for Altera and Xilinx, as the fabrication of FPGAs was being outsourced and meant that no specialist technology interaction with the fabrication companies was needed. Indeed, it is worth noticing that silicon manufacturers now see FPGA technologies as the most advanced technology to test their fabrication facilities.

Programmable resource functionality A number of different offerings again exist in terms of the basic logic block building resource used to construct systems. Companies such as Algotronix, Crosspoint and Plessey had offered FPGAs which were fine-grained with simple logic gates or multiplexers, being offered as the logic resources. With interconnect playing an increasing role in determining system performance, these devices were doomed, as described in Chapter 1. There also existed a number of options in the coarser-grained technologies, namely the PLD-type structure or the LUT. The PLD structure was related to logic implementation whereas the LUT was much more flexible and it was a concept understood by computer programmers and engineers. Examining the current FPGA offerings, it is clear to see that the LUT-based structure now dominates with the only recent evolution an increase in the size of the LUT from 4-input to 5/6-input in the Xilinx Virtex™-5 technology and to 6-input in the Altera Stratix®III family.

Change of FPGA market With the FPGAs growing in complexity, it now meant that the FPGA had gone from being primarily a glue logic component, to being a major component in a complex system with DSP being the target area of this book. However, it should still be observed that the FPGA is an important part of the telecommunications industry. This means that FPGA vendors have to compare their technology offerings in terms of new competitors, primarily DSP processor developers such as TI, Analog Devices and multi-core developers. Some of these technologies were presented in the previous chapter.

Tool flow Initially, FPGAs were not that complex, so up until the mid 1990s, it was usual that the designer would perform manual placement of designs. The first major tool development was automatic *place and route* tools which still plays a major role in the vendors' tool flow. However, increasingly there is a well-recognized need for system-level design tools, to address latest design challenges. For this reason, FPGA vendors have been increasingly involved in developing system-level design tools such as the DSP and SOPC builder from Altera, and System Generator for DSP and AccelDSP™ from Xilinx in addition to system-level offerings from tools vendors. This is an increasing problematic issue as tools tend to lag well behind technology developments and is a major area of focus in this book.

It has now got to the stage that FPGAs represent system platforms. This is recognized by both major vendors who now describe their technology in these terms; in Xilinx's case, they describe their Virtex™ as a *platform* FPGA and with Altera, they describe their Stratix III as a *high end* being able to design *entire systems-on-a-chip*. Thus we have moved from the era of programmable cells connected by programmable interconnect as highlighted at the start if this section, to devices that are complex, programmable SoCs which comprise a number of key components, namely dedicated DSP processor blocks, soft and hard processor engines.

5.3 Altera FPGA Technologies

Altera is one of the two main FPGA companies and evolved their initial architectures, based on the PLD structure, described in the previous section. Its current FPGA portfolio is organized into several different technologies, as outlined in Table 5.2. Altera FPL families are organized

Table 5.2 Altera's FPGA family range

Type	Family	Brief description
CPLDs	MAX®II	Technology with numerous interconnected PLD-based blocks. Family includes MAX®II, MAX®3000A and MAX®7000
FPGAs	Cyclone	Cost-optimized, memory-rich FPGA family
FPGAs	Stratix®	High-performance, low-power FPGAs
FPGAs	Stratix® GX	FPGA with high-speed serial transceivers with a scalable, high-performance logic array
Structured ASIC	HardCopy®	Low-cost, high-performance structured ASIC with pin-outs, densities, and architecture that complement Stratix II devices

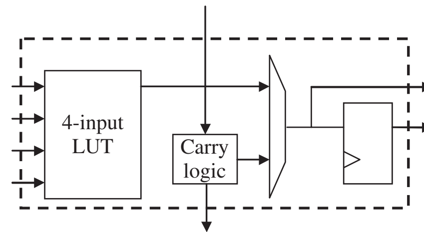


Figure 5.3 Block diagram of Altera LE cell

into: configurable programmable logic devices (CPLDs) comprising the MAX® and MAX®II series families; low-cost FPGA families such as the Cyclone and Cyclone II families; high-density FPGA families such as the Stratix®, Stratix®II, Stratix®III and Stratix®GX families and; structured ASIC solutions HardCopy® and HardCopy®II families. The section will concentrate mostly on the Stratix®III family, as we are targeting DSP and particularly, high-performance DSP applications; however the MAX7000 series FPGA is also briefly reviewed as it is an obvious extension of the PLD concept.

The core block in the Altera FPGAs has been the *logic element* (LE) which is given in Figure 5.3. This is very similar to the Xilinx's *logic cell* (LE) in their XC4000 and early Virtex™ FPGA families, although Xilinx have migrated recently to a 6-input LUT. The cell was built from the concept of meeting the criteria for implementing a purely combinational logic function (LUT table only), delay or shift function (flip-flop only) or sequential logic circuit (combinational logic feeding into flip-flops). Thus all configurations are provided along with various multiplexing and interconnection. The notion of choosing a 4-input LUT (rather than larger or smaller LUT) probably dates back to the work by Rose *et al*. (1990) which showed that this size of LUT produced the best area efficiency for a number of different examples. Combinational logic implementations can then be constructed using series of these 4-input LUTs by using the programmable interconnect to link these 4-input LUTs together to build larger LUTs. The cell also provides a fast carry logic circuit for accelerating the implementation of adders using the approach, illustrated in Figure 3.3(b).

5.3.1 MAX® 7000 FPGA Technology

The basic PLD structure is given in Figure 5.4 where each of the PLD blocks are of the form given in detail in Figure 5.5. The framework provides a mechanism for connecting PLD blocks together via the programmable interconnect, in the same way that several PLD chips would have been connected to provide blocks of complex logic using commercial PLDs, with the added advantage that the PLD-based FPGA interconnect is programmable, the importance of which has already been highlighted on a number of occasions.

The MAX® and MAX®II series families from Altera (Altera Corp. 2000) are extensions of the basic concept of PLD technology. The MAX 7000 architecture is given in Figure 5.6 and consists of logic array blocks (LABs) which comprise 16-macrocell arrays, a programmable interconnect array (PIA) to allow connection of the blocks to each other and connection of various control inputs such as clock, reset etc., and I/O control blocks to allow internal interfacing to both the LABs and PIA. Four dedicated inputs allow high-speed, global control signals such as clock, reset and enable, to be fed into each macrocell and I/O pin. Each LAB is fed by 36 general logic signals from the

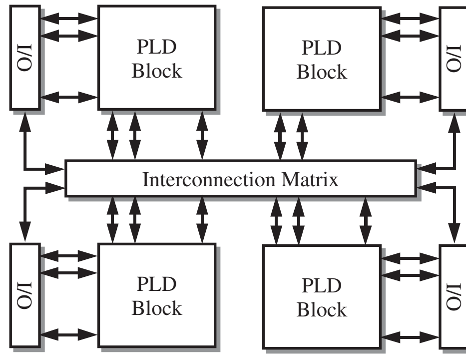


Figure 5.4 Generalized PLD-based FPGA architecture

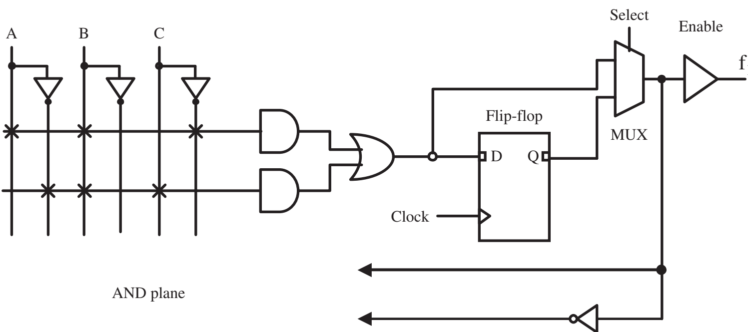


Figure 5.5 PLD building block

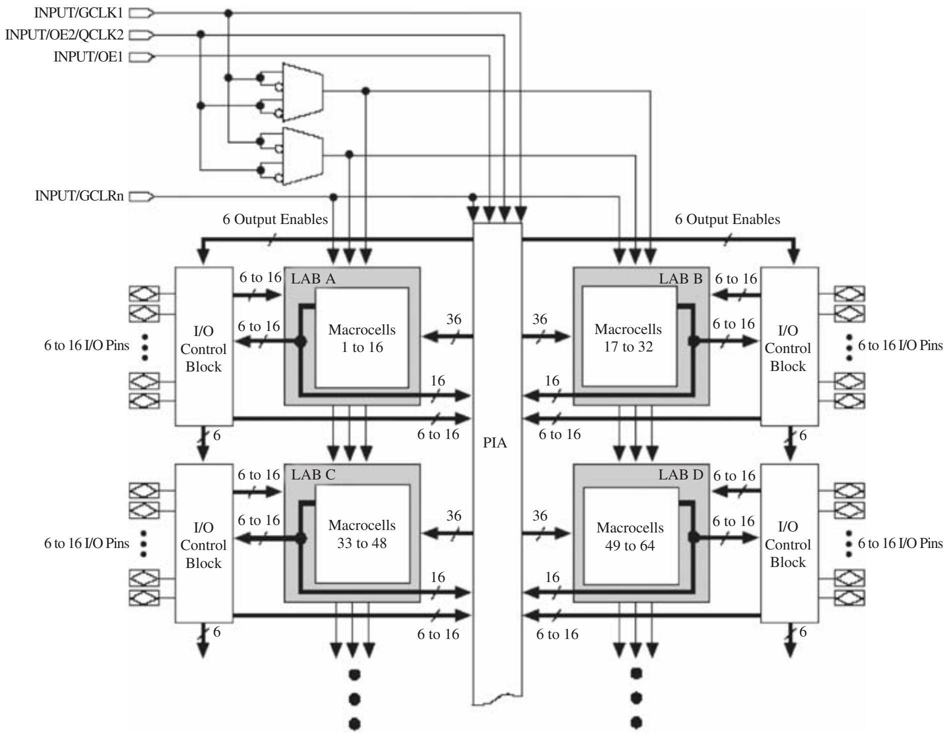


Figure 5.6 Altera MAX 7000E and MAX 7000S device block diagram. Reproduced by permission of Altera Corp.

PIA, global controls for use with the register part of the functionality and also direct connection to I/O pins to the registers in order to minimize delays for off-chip and on-chip communication.

The key computational part is the MAX 7000 macrocell given in Figure 5.7, which can be configured for either sequential or combinatorial logic operation. It consists of a logic array, a product-term select matrix and a programmable register. The logic array allows five product terms to be summed and product-term select matrix allows this to be used as the main (primary) output of the cell per macrocell, thereby providing a combinational output, or as part of a larger logic function, i.e. a secondary input to the register, allowing sequential logic circuits to be implemented. The flip-flop can be used as a delay, or part of a larger sequential circuit, and can be controlled in a number of ways in terms of reset or clock, using the various select functions. The cell also contains shareable expanders which allow inverted product terms to be fed back into the logic array, and parallel expanders to allow creation of larger fan-in logic functions by allowing inputs from other macrocells.

Logic is routed between LABs via the programmable interconnect array (PIA), which is a programmable path and allows any signal to connect to any destination on the device. The route is created by programming an E^2PROM cell which controls one input of a 2-input AND gate thereby disabling or enabling the connection. One of the key advantages of the PLD is preserved in the PLD-based FPGA, namely the routing delays are fixed unlike other FPGA architectures described later, which are cumulative, variable, and path-dependent and can cause problems when achieving timing closure (timing closure is the process when all of the individual delays in the design have

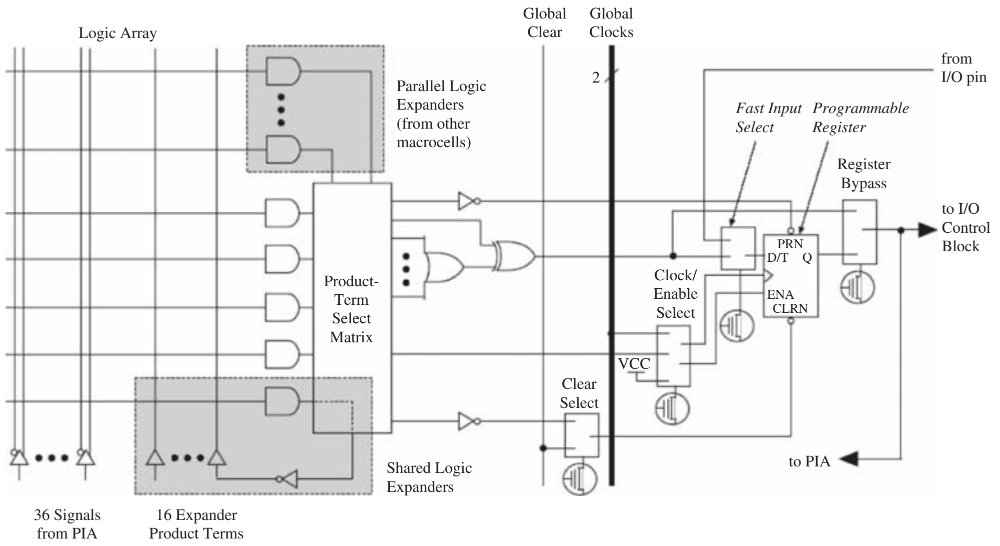


Figure 5.7 LAB block diagram. Reproduced by permission of Altera Corp.

to meet the various timing constraints namely, critical path and edge to edge timing delays). Thus this makes the timing performance easier to predict. From a DSP perspective, these devices have limited usage as the main combinational blocks are for conventional logic implementations.

5.3.2 *Stratix® III FPGA Family*

A number of variations exist for the Stratix® III FPGA family. The Stratix III E family would seem to be targeted toward DSP applications given the memory and multiplier rich for data-centric applications. The Stratix III E family offers a number of features useful for DSP applications, including: 48 000–338 000 equivalent LEs, up to 20 Mbits of memory and a number of high-speed DSP blocks that provide dedicated implementation of multipliers, multiply accumulate blocks and FIR filter functionality. In addition, the devices also provide adjustable voltage levels, a number of PLLs and various clocks. The floorplan of the Altera Stratix EP3SE50 is given in Figure 5.8.

Adaptive Logic Modules

In the Stratix III, the LE concept has been extended, leading to what Altera term an *adaptive logic module* or ALM, as shown in Figure 5.9. The ALM is based on 8-input LUT which can be *fractured*, allowing the original LE configuration of Figure 5.3; however, it also allows a number of other combinations, including not surprisingly, 7-input and 6-input LUTs, but also combinations of 5-input and 3-input, and even 5-input and 5-input, 4-input and 5-input, and 6-input and 6-input LUTs (as long as the total number of individual inputs does not exceed 8!). In addition, there are two dedicated adders and two registers. The 2:1 register-to-LUT ratio in ALMs ensures that the FPGA is not register-limited. The two adders can perform a 2-bit addition or a single ternary addition. The core concept of a LUT–multiplexer–register combination still remains, but just with a bigger LUT.

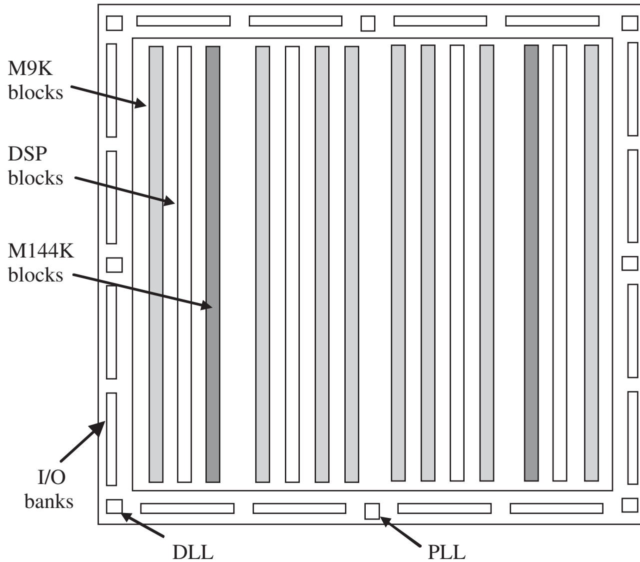


Figure 5.8 Altera Stratix EP3SE50 floorplan

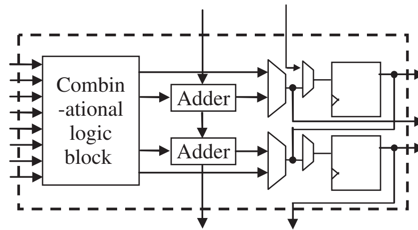


Figure 5.9 Altera ALM block diagram

Memory Organization

The Altera Stratix FPGA has a hierarchy of memory called TriMatrix memory, ranging from smaller, distributed memory blocks right through to larger memory blocks which provide a memory capacity of 17 Mbits, performing at rates of over 600 MHz. The types of memory are listed in Table 5.3. Three types are included and listed below:

MLAB blocks or memory LABs which are created from the ALMs and is a new derivative of the LAB. MLAB is a superset of the LAB and can give 640 bits of simple dual-port SRAM. As each ALM can be configured as either a 64 × 1 or 32 × 2 block, the MLAB can be configured as a 64 × 10-bit or 32 × 20-bit simple dual-port SRAM block. MLAB and LAB blocks co-exist as pairs, allowing 50% of the LABs to be traded for MLABs. The MLABs would tend to be used as localized memory in DSP applications to store temporary and local data, thus giving high performance as a lot of the memory can be accessed in parallel.

Table 5.3 Stratix memory types and usage

Memory type	Bits/block	No. of blocks	Suggested usage
MLAB	640	6750	Shift registers, small FIFO buffers, filter FIFO buffers, filter delay lines
M9K	9,216	1144	General-purpose applications, packet headers, cell buffers
M144K	147,456	48	Processor code storage, packet or video frame buffers

M9K blocks which are 9 kB block RAM allow storing of fairly large data. These are located as shown in Figure 5.8.

M144K blocks are larger, 144 kB of RAM, and from a DSP perspective for example, could be used in image processing applications to store images.

Each embedded memory block can be independently configured to be a single- or dual-port RAM, ROM, or shift register

DSP Processing Blocks

A key component of the Altera Stratix III FPGA, is the DSP function block; the largest Stratix device supports up to 112 such blocks, operating up to 550 MHz, thereby allowing huge performance capability for many DSP applications. The detailed block diagram for a half-DSP block is shown in Figure 5.10. The input wordlength to the input register block is 144 bits which is split into 8 of 18-bit words for the multiplier inputs shown as *dataa* and *datab* respectively; the output is 72 bits. The data is registered into and out of the DSP block, thereby avoiding any timing problems in meeting critical path delays, when the DSP block forms part of a larger system. In addition to these register inputs, the DSP blocks supports optional pipelining for higher speed, as indicated by the *pipeline register bank* in the figure. As will be seen in later chapters, the delays introduced by the pipeline stages, have to be taken into account in the development of the circuit architecture for the DSP function under consideration.

The first stage of the block comprises two dual multiplication/accumulation blocks which has been clearly developed to support specific DSP functions, namely: a 2-tap FIR filter configuration for each block, with the second stage adder/accumulator after the optional pipeline being used to sum the two 2-tap filters to give a four tap filter within the block; a part of a FFT bufferfly stage and; complex operations such as a complex multiplication, where the input stage ideally implements the complex multiplication of $a + jb$ by $c + jd$ given as $(ac - bd) + j(ad + bc)$. The multipliers are 18-bit but can also function as two 9-bit multipliers. From the Altera literature (Altera Corp. 2007), it is also indicated that 9-bit, 12-bit, 18-bit and 36-bit word lengths are supported as well as signed and unsigned.

This multiply accumulate stage is then followed by an optional pipeline register bank and then the second stage adder/accumulator. Once again, the adder is configured to give maximum advantage to implement fast DSP systems, by providing a loop back from the output register bank allowing recurrence functions, commonly found in common in IIR filter implementations, to be computed. In addition, the DSP block provides the mechanism via the *chainout adder*, to add in the output from the DSP block above; this is possible as the DSP blocks are connected in columns as shown in Figure 5.8. This means it is now possible to implement a 4-tap filter by connecting two DSP blocks together in this way, and even much larger filters, with accordingly more DSP blocks.

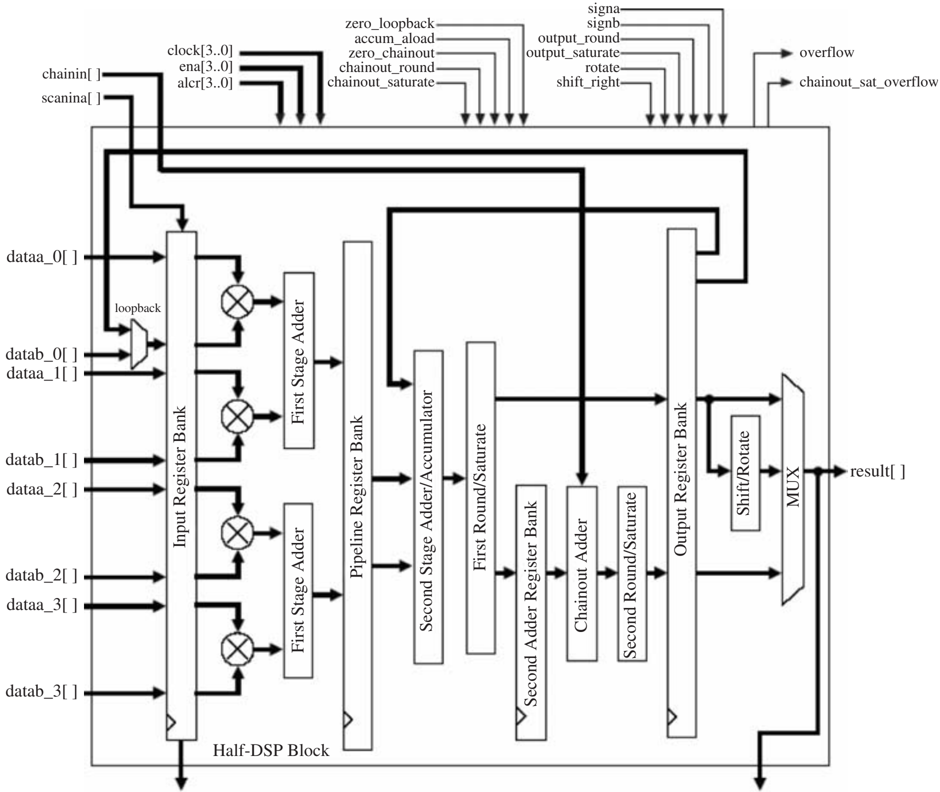


Figure 5.10 Altera Stratix DSP block diagram (Altera Corp. 2007). Reproduced by permission of Altera Corp.

Rounding and saturation blocks are included after the *second stage adder/accumulator* and *chainout adder* blocks. The word growth is predictable in the first stage, but given that the output can be continually fed back via the loop back, it is essential to employ rounding and/or saturation to avoid overflow; likewise for the *chainout adder* when very larger filters are being created. The reasons for this have been highlighted in Chapter 4.

In summary, the DSP block can perform five basic DSP operations, as illustrated in Table 5.4 which is a summary of the information presented in the data sheet. This gives more detail on the arithmetic (signed/unsigned) and whether rounding can be applied. Two rounding modes are supported in namely *round-to-nearest-integer* which is normal rounding in DSP systems and *round-to-nearest-even* mode which as the name suggest rounds to nearest even number. Two saturation modes are supported, namely *asymmetric* and *symmetric* saturation. In 2's complement format, the maximum negative number that can be represented is -2^{n-1}, i.e. -128 for 8-bit, while the maximum positive number is $2^{n-1} - 1$, i.e. 127 for 8-bit. This is the range to which any number will be saturated in the asymmetric case, where the range is $-2^n + 1$ i.e. -127 to $2^{n-1} - 1$, i.e. 127 in the symmetrical case. There are 16 different cases for rounding and saturation in the 44-bit representation, thereby allowing a trade-off between accuracy and dynamic range, depending on the application.

Table 5.4 DSP block operation modes(Altera Corp. 2007)

Mode	Multiplier width (bits)	No. of mults	per block
Independent multiplier	9	1	8
	12	1	6
	18	1	4
	36	1	2
	Double	1	2
Two-multiplier adder	18	2	4
Four-multiplier adder	18	4	2
Multiply–accumulate	18	4	2
Shift	36	1	2

Stratix Clock Networks and PLLs

Whilst it is important to understand the DSP blocks in detail for the FPGA technology under consideration, it is also important to have some appreciation of the clocking strategies and it can be critical in obtaining the required performance. The Stratix III devices have a number of dedicated global clock networks (*GCLKs*), regional clock networks (*RCLKs*), and periphery clock networks (*PCLKs*). These are organized into a hierarchical clock structure that provides up to 220 unique clock domains (16 GCLK + 88 RCLK + 116 PCLK). As the clock network can consume a considerable amount of power, the Quartus® II software compiler, automatically turns off clock networks not used in the design. The Stratix has up to 12 PLLs per device and up to 10 outputs per PLL, each of which can be programmed independently, allowing the creation of a customizable clock frequency, with no fixed relation to any other input or output clock.

In all, 16 GCLKs are provided which seem to be organized in an H clock tree network such as that shown in Figure 5.11. This provides an equal delay to each clock signal thereby balancing the skew. The GCLKs signals can drive functional blocks such as ALMs, DSP blocks, TriMatrix memory blocks and PLLs. Stratix III device I/O elements (IOEs) and internal logic can also drive GCLKs, to create internally generated global clocks and other high fan-out control signals such as globals resets or clock enables. The RCLK networks only pertain to the quadrant they drive into, and once again, can be used for globals resets or clock enables. Periphery clock (PCLK) networks are a collection of individual clock networks driven from the periphery of the Stratix III device.

Alternatively, the PLL can be used to synchronize the phase and frequency of an internal or external clock, f_{co} to an input reference clock, f_{ci}. The basic diagram for a PLL of the type used in the Altera Stratix, is given in Figure 5.12. The *voltage-controlled oscillator* (VCO) generates a periodic output signal. If we assume that the oscillator starts at nearly the same frequency as the reference signal, then if the VCO falls behind that of the reference clock f_{ci}, this will be detected by a phase detector block called a *phase-frequency detector* (PFD). It will do this by generating an up (U) or down (D) signal that effectively determines whether the VCO needs to operate at a higher or lower frequency. These signals will then cause a circuit called a *charge pump* to change the control voltage, so that the VCO either speeds up or slows down. The *low-pass filter* sometimes called a *loop filter*, smooths out the abrupt control inputs from the charge pump, thereby preventing voltage overshoot and converts these U and D signals to a voltage that is used to bias the VCO via the *bias generator*, and thus determine how fast the VCO operates. A divider counter ($\div N$) in the feedback loop increases the VCO frequency, given as f_{co} above the input reference frequency, meaning that the feedback clock to the PFD from the VCO is N times that of the input reference

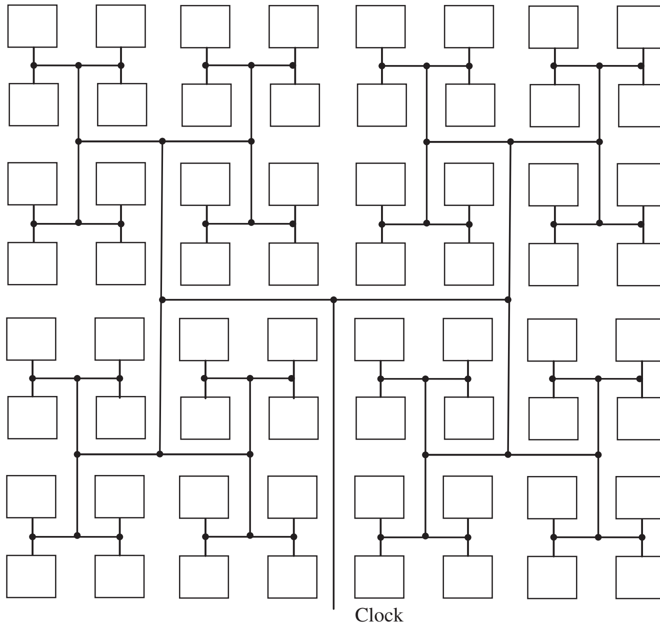


Figure 5.11 H tree buffer for clock

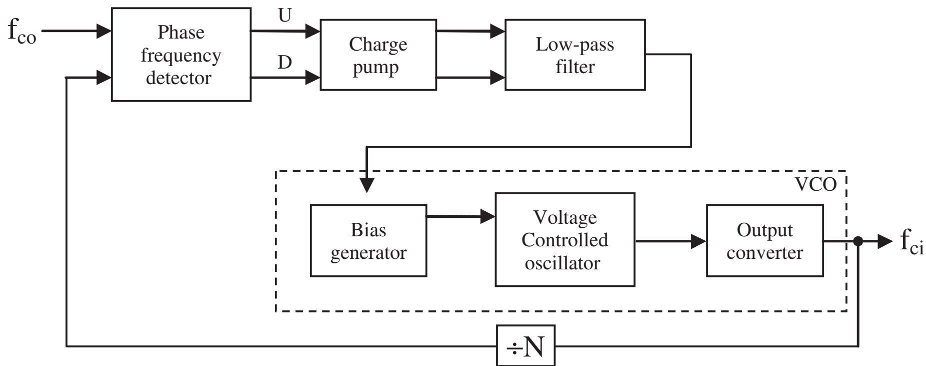


Figure 5.12 Phase lock loop

clock f_{ci}. Thus, the feedback clock applied to one input of the PFD is locked to the clock, f_{ci} that is applied to the other input of the PFD.

In the Altera Stratix FPGA, the VCO output from left/right PLLs in the FPGA die, feeds 7 post-scale counters, whilst the corresponding VCO output from top/bottom PLLs can feed 10 post-scale counters. These counters then allow the generation of a number of harmonically related frequencies to be produced by the PLL.

Table 5.5 Typical memory interface speeds for Altera Stratix (Altera Corp. 2007)

Memory interface standard	I/O standard	Max. clock rate	
		−4 speed grade	−2 speed grade
DDR SDRAM	SSTL-2	333	400
DDR2 SDRAM	SSTL-1.8	267	400
DDR3 SDRAM	SSTL-1.5	200	200
RLDRAM II	1.8V HSTL	250	350
QDRII SRAM	1.8V HSTL	250	350
QDRII+ SRAM	1.5V HSTL	250	400

I/O and External Memory Interfaces

The Stratix III FPGA produces a number of standard interfaces to each of the low-voltage CMOS and TTL standard programmable input and output pins. These pins are contained within a complex *I/O element* (IOE), located in I/O blocks around the Stratix device periphery; this contains bidirectional I/O buffers and I/O registers, allowing the pin to be configured for complete embedded bidirectional *single* or *double data rate* (DDR) transfer where both the rising and falling edges of the clock are used to accelerate data transfer. As indicated on Figure 5.8, there are up to four IOEs per row I/O block and four IOEs per column I/O block; the row IOEs drive the row, column, or direct link interconnects and the column IOEs, drive the column interconnects. A number of standard features are supported, including programmable input and output delay, slew rate, bus-hold and pull-up resistors, open-drain output as well as a number of on-chip series termination modes. This I/O configuration allows a 132 full duplex 1.25 Gbps true *low-voltage differential signaling* (LVDS) channels (132 Tx + 132 Rx) to be supported on the row I/O banks.

The I/O structure also provides support for high-performance external memory standards; DDR memory standards are supported such as DDR3, DDR2 and DDR SDRAM, QDRII+ and QDRII SRAM and RLDRAM II at frequencies of up to 400 MHz. A sample of some of the data rates for different speed grade technologies is listed in Table 5.5

Gigabit Transceivers

The Stratix® III device family offers a series of high-speed Gigagbit transceiver blocks, which allow data to be transferred at high speed, between different system devices in the DSP system, i.e. from chip to chip. The transceiver uses one pair of differential signals, i.e. a pair of signals which always carry opposite logical values, to *transmit* data and another set, to *receive* the data; hence the transmit and receive properties lead to the name *transceiver*. These transceivers operate at very high data rates, in the case of the Stratix® III device family at up to 1.25 Gbps and support a number of communication protocols such as Utopia and Rapid I/O™. These are explained below.

Conventional systems can be constructed by connecting systems devices using a hierarchy of buses; devices are thus placed at the appropriate level in the hierarchy, according to the performance level they require, i.e. low-performance devices placed at lower-performance buses, etc. A number of specific techniques have been introduced to achieve the performance requirements of individual connections, such as increasing bus frequency or width, splitting the transactions and allowing out-of-order completion. This required the development of individual system interfaces and complicated the design process. Over the past several years, with the development of the concept of the shared multi-drop bus that allows the full range of low/high bandwidth, high-speed communications has grown in interest. The Rapid I/O™ standard facilitates the operation of such a platform and

effectively sits on top of the high speed gigagbit transceiver just described. In operation, a master or initiator processor generates a request transaction, which is transmitted to a target over the high-speed communications framework. The target then generates a response transaction back to the initiator to complete the operation. The Rapid I/O™ transactions are encapsulated in packets, which include all of the necessary bit fields to ensure reliable delivery to the targeted end point. Rapid I/O™ provides the same programming models, addressing mechanisms and transactions for both serial and parallel implementations, including basic memory mapped I/O transactions, port-based message passing and globally shared distributed memory with hardware-based coherency (Bouvier 2007). It can also manage any resulting errors that occur as each packet includes an end-to-end cyclic redundancy check (CRC). The adoption of Utopia and Rapid I/O™ thus reduces the design complexity by providing a standard interface for communications.

Device Security

Security is a major concern in FPGA technology as most of the major FPGA devices are based on SRAM technology; like standard memory, the contents can be easily read. Typically the designer will create a design using the commercial vendors' proprietary software, resulting in a configuration data file which programs the SRAM-based device. An EPROM can then be used to store the FPGA programming information, or more commonly it can be stored in system memory of an available microprocessor and loaded at power-up. Thus, the configuration information data can be captured, either from the EPROM or the FPGA SRAM configuration data locations in the FPGA which is a problem as it represents the designer's intellectual property.

A feature has thus been included in the Stratix III devices that allows the FPGA configuration bitstream to be encrypted using the industry standard, AES algorithm. The AES algorithm works on the principle of a *security key* stored in the Stratix III device, which is used to encrypt the configuration file. The design security feature is available when configuring Stratix III devices, using the fast, passive, parallel (FPP) configuration mode with an external host such as a microprocessor, or when using fast active serial or passive serial configuration schemes (Altera Corp. 2007). The design security feature is also available in remote update with fast active serial configuration mode.

5.3.3 Hardcopy® Structured ASIC Family

These devices are structured ASICs with pin-outs, densities, and architecture that complement Stratix® II devices. The main focus of the HardCopy® device is to strip the reprogrammable FPGA logic, routing, memory, and FPGA configuration-related logic and replace SRAM configuration resources by direct metal connections. Thus, it is envisaged that the designer would prototype the design using the Stratix® II FPGA family and then implement the volume product using HardCopy®.

The memory, clock networks and PLLs are identical in both devices as these are standard components, but the Stratix® II adaptive logic modules (ALMs) and dedicated DSP blocks are replaced by combinations of logic blocks, known as HCells. The Quartus II software used to implement the design, then uses the library of pre-characterized HCell macros to replace Stratix II ALM and DSP configurations before the design is transferred to the FPGA. This is achieved by having eight HCell macros which implement the eight supported modes of operation for the Stratix II DSP block for various forms of 9, 18 and 36-bit multiplication, multiply–accumulate and complex multiplication and addition.

The HardCopy® II memory blocks can also implement various types of memory, with or without parity, including true dual-port, simple dual-port, and single-port RAM, ROM, and FIFO buffers.

HardCopy II devices support the same memory functions and features as Stratix II FPGAs, specifically the 4 k M4K RAM blocks and the 512-bit M-RAM blocks.

Of course, one the attractions of the structured ASICs compared with conventional SRAM-based FPGA implementations, is that they do not need to be programmed as the programmability has been effectively removed. Thus, some features need to be turned off such as the design security feature needed to encrypt the data stream and the configuration status pins. The HardCopy II structured ASIC follows the same principle of ASIC power-up except that is has an instant *on* time delay of 50 ms. During this time, all registers are reset; having resettable flip-flops is highly attractive feature as, for cost reasons, not all registers will be made to be resettable in ASIC implementations.

5.4 Xilinx FPGA Technologies

The first FPGA was the Xilinx XC2000 family developed in 1982. The basic concept was to have programmable cells, connected to programmable fabric which in turn were fed by programmable I/O as illustrated by Figure 5.13. This differentiated Xilinx FPGAs from the early Altera devices which were PLD-based; thus the Altera FPGAs did not possess the same high levels of programmable interconnect. The architecture comprised cells called *logic cells* or LCs which had functionality very similar to the Altera LE given earlier in Figure 5.3. The interconnect was programmable and was based on the 6-transistor SRAM cell given in Figure 5.14. By locating the cell at interconnections, it could then provide flexible routing by allowing horizontal-to-horizontal, vertical-to-vertical, vertical-to-horizontal and horizontal-to-vertical routing, to be achieved. The I/O cell had a number of configurations that allowed pins to be configured as input, output and bidirectional, with a number of interface modes.

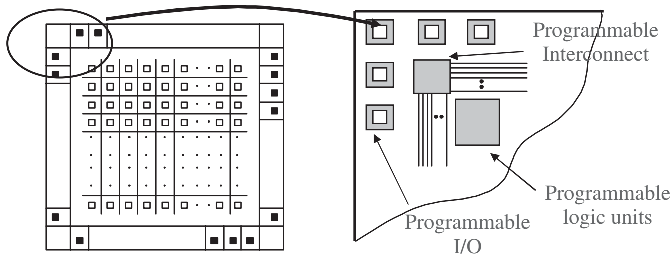


Figure 5.13 Early Xilinx FPGA architecture

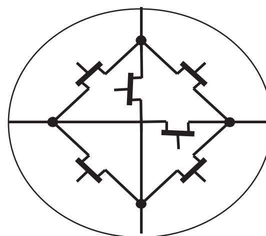


Figure 5.14 Xilinx FPGA SRAM interconnect

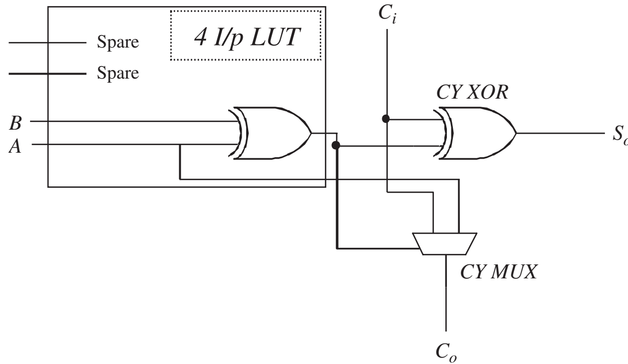


Figure 5.15 Adder implementation on Xilinx Virtex™ FPGA slice

At this stage, FPGAs were viewed as glue logic devices with Moore's law providing a continual expansion in terms of logic density and speed. The device architecture continued largely unchanged from the XC2000 right up to the XC4000; for example, the same LUT table size was used. The main evolution was the inclusion of the fast adder where manufacturers observed that, by including an additional multiplexer in the LE cell, a fast adder implementation could be achieved by mapping some of the logic into the fast carry adder logic, and some into the LUT. The principle is illustrated for the Virtex™ FPGA device in Figure 5.15. At this stage, the device was still being considered as glue logic for larger systems, but the addition of the fast adder logic started to open up the possibility of implementing a limited range of DSP systems, particularly those where multiplicative properties were required, but which did not require the full range of multiplicands. This formed the basis for a lot of early FPGA-based DSP implementation techniques which are described in Chapter 6.

At that time, a lot of FPGA products manufacturers faded away and there began a period defined as accumulation (see Table 1.1) where FPGAs started to accumulate more complex components, starting with on-board dedicated multipliers, which appeared in the first Xilinx Virtex™ FPGA family (Figure 5.16), Power-PC blocks and gigabit transceivers with the Xilinx Virtex™-II pro and Ethernet MAC with the Virtex™-4. As with the Altera technology, it can be seen from Figure 5.16, that the Xilinx FPGA was now becoming increasingly like a SoC with the main aim of the programmability to allow the connection together of complex processing blocks with the LCs used to implement basic logic functionality. The fabric now comprised the standard series of LCs, allowing functions to be connected as before, but now complex processing blocks such as 18-bit multipliers and PowerPC processors (Figure 5.17), were becoming commonplace. The concept of *platform FPGA* was now being used to describe recent FPGA devices to reflect this trend. The full current FPGA family available from Xilinx is given in Table 5.6.

5.4.1 Xilinx Virtex™-5 FPGA Technologies

The text concentrates on the latest Virtex™ FPGA, namely the Virtex™-5 family on the basis that it represents a more evolved member of the FPGA family. Description of the CPLD family is not included on the basis that details are included on the Xilinx web pages. The Virtex™-5 comes in a variety of flavours, namely the *LX* which has been optimized for high-performance logic, the *LXT* which has been optimized for high-performance logic with low-power serial connectivity, and the

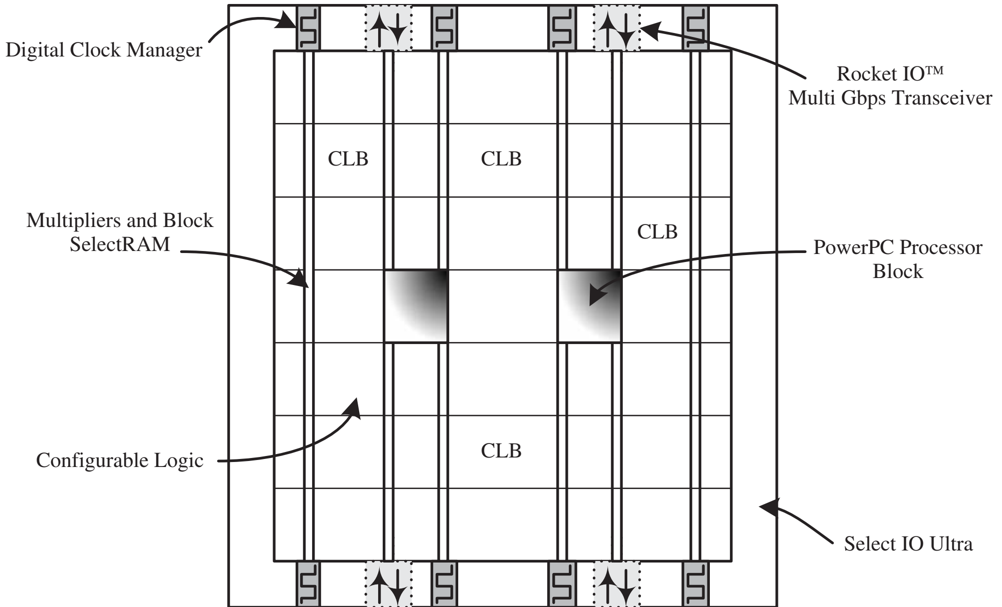


Figure 5.16 Virtex™-II Pro FPGA architecture overview

Table 5.6 Xilinx's FPGA family range

Type	Family	Brief description
CPLDs	XC9500XL	Older CPLD technology
CPLDs	CoolRunner	High-performance, low-power CPLD
FPGAs	Virtex™ / E / EM	Main Xilinx high-performance FPGA technology.
FPGAs	Spartan	Low-cost, high-volume FPGA

SXT which has been optimized for DSP and memory-intensive applications with low-power serial connectivity. The Xilinx Virtex™-5 family has a two speed-grade performance gain and is able to be clocked at 550 MHz. It has a number of on-board IP blocks and a number of DSP48E slices which give a maximum of 352 GMACS performance. It also provides up to 600 pins, giving an I/O of 1.25 Gbps LVDS and, if required, RocketIO GTP transceivers which deliver between 100 Mbps and 3.2 Gbps of serial connectivity. It also includes hardened PCI Express endpoint blocks and Tri-mode Ethernet MACs.

Virtex™-5 Configurable Logic Block

The logic implementation in the Xilinx device is contained within *configurable logic blocks* or CLBs. Each CLB is connected to a switch matrix for access to the general routing matrix as shown in Figure 5.18 and contains a pair of slices which are organized into columns, each with an independent carry chain. For each CLB, slices in the bottom of the CLB are labelled as SLICE(0),

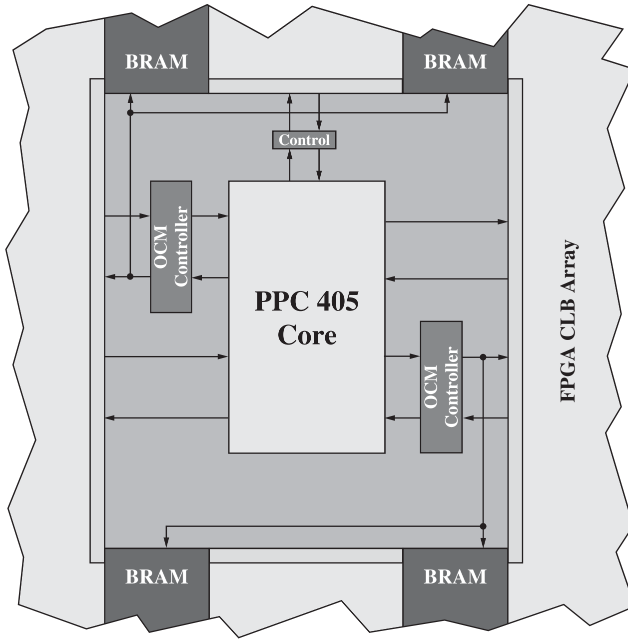


Figure 5.17 PowerPC processor block architecture

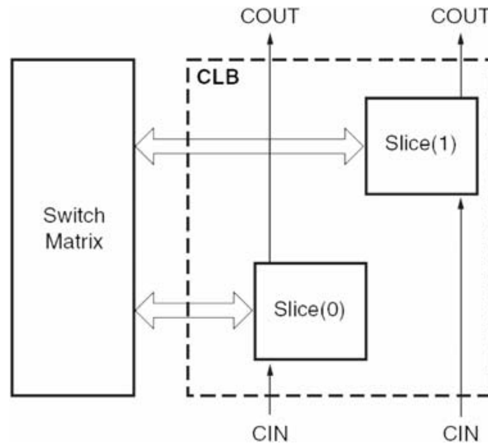


Figure 5.18 Arrangement of slices within the CLB

and slices in the top of the CLB, are labelled as SLICE(1) and so on. Every slice contains four logic-function generators (or LUTs), four storage elements, wide-function multiplexers, and carry logic and so can be considered to contain four of the logic cell logic as given in Figure 5.19. In addition to this, some slices, called SLICEM, support two additional functions: storing data using distributed RAM and shifting data with 32-bit registers.

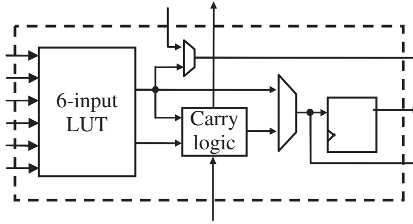


Figure 5.19 Logic cell functionality within a slice

The basic logic cell configuration is similar to the Altera LE given of Figure 5.3. It comprises a logic resource in this case, a 6-input LUT connected to a single flip-flop, via a number of multiplexers, together with a circuit for performing fast addition. As with the Altera LE element, the basic logic cell has been designed to cope with the implementation of combinational and sequential logic implementations, along with some simple DSP circuits that use an adder. The basic combination of LUT plus register has stayed with the Xilinx architecture, and has now been extended from a 4-input LUT in the Xilinx XC4000 series and Virtex™-5 series FPGA family to a 6-input LUT; this is a reflection of improving technology as governed by Moore's law. It is now argued in Xilinx Inc. (2007a) that a 6-input rather than a 4-input LUT which went all the way back to the study by Rose *et al.* (1990), now provides a better return on silicon area utilization for the critical path needed within the design. The combination of LUTs, flip-flops (FFs), and special functions such as carry chains and dedicated multiplexers, together with the ways by which these elements are connected, has been termed *ExpressFabric* technology.

The CLB can implement the following: a pure logic function by using the 6-input LUT logic and using the multiplexers to bypass the register; a single register using the multiplexers to feed data directly into and out of the register; and sequential logic circuits using the LUTs feeding into the registers. Scope is also provided to create larger combinational and sequential circuits, using the multiplexers to create large LUTs and registers. One special feature of the 6-input LUT is that it has two outputs. This allow the LUT to implement two arbitrarily defined, five-input Boolean functions, as long as these two functions share common inputs (see Figure 5.20). This is an attempt to provide better utilization of the LUT resource when the number of inputs is smaller than six. This concept also allows the logic cell to implement a full adder, as shown in Figure 5.15 whilst at the same time, using the additional inputs and outputs to realize a 4-input LUT for some other function. This provides better utilization of the hardware in many DSP applications, where otherwise LUTs would be *wasted* to just provide a single gate implementation for a adder.

As with the Altera technology, the register resource is also very flexible, allowing a wide range of storage possibilities ranging from edge-triggered D-type flip-flops to level-sensitive latches, all with a variety of synchronous and asynchronous inputs for clocks, clock enables, set/reset. The D input can be driven directly from a number of sources, including the LUT output, other D-type flip-flops and external inputs.

One of the advantages of the larger LUT in the Xilinx Virtex™-5 device is that it provides larger distributed RAM blocks and SRL chains. A sample of the various distributed memory configurations is given in Table 5.7 which gives the number of LUTs needed to create the various memory configurations listed. The distributed RAM modules have synchronous write resources, and can be made to have a synchronous read by using the flip-flop of the same slice. By decreasing the *clock-to-out* delay, this will improve the critical path, but adds an additional clock cycle latency.

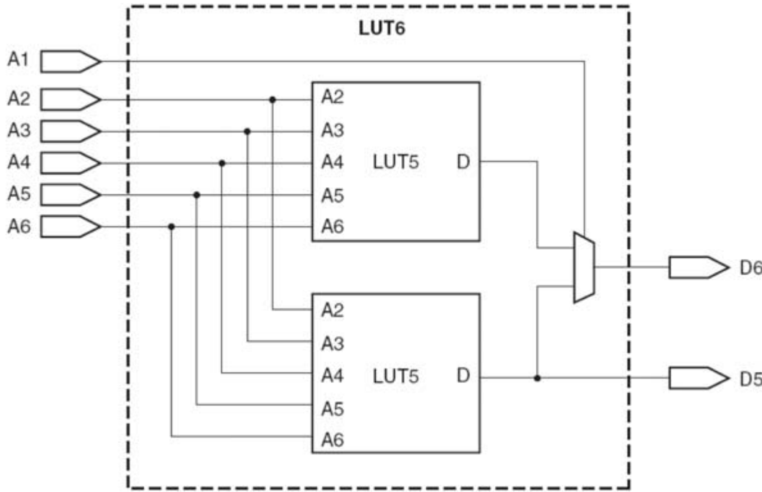


Figure 5.20 Arrangement of slices within the CLB

Table 5.7 Number of LUTs for various memory configurations

Memory type	No. of memory locations 32	64	128
Single-port	1 (1-bit)	1 (1-bit)	2 (1-bit)
Dual-port	2 (1-bit)	2 (1-bit)	4 (1-bit)
Quad-port	4 (2-bit)	4 (2-bit)	
Simple dual-port	4 (6-bit)	4 (3-bit)	

A number of memory configurations have been listed. For the single-port configuration, a common address port is used for synchronous writes and asynchronous reads. For the dual-port configuration, the distributed RAM has one port for synchronous writes and asynchronous reads, which is connected to one function generator and another port for asynchronous reads, which is connected to a second function generator. In simple dual-port configuration, there is no read from the write port. In the quad-port configurations, the concept is expanded by creating three ports for asynchronous reads, and three function generators plus one port for synchronous writes and asynchronous reads, giving a total of four functional generators.

The consideration of larger memory blocks is considered in the next section, but the combination of smaller distributed RAM, along with larger RAM blocks, provides the same memory hierarchy concept that was purported by the Altera FPGA, admittedly in different proportions. The LUT can also provide a ROM capability, and as Chapter 6 will illustrate, the development of programmable shift registers. The Virtex™-5 function generators and associated multiplexers some of which were highlighted in Figure 5.19, can implement one 4:1 multiplexers using one LUT, one 8:1 multiplexers using two LUTs etc.

Table 5.8 Virtex™-5 memory types and usage

Memory type	Bits/block	No. of blocks	Suggested usage
Distributed RAM/slice	1024	14720	Shift registers, small FIFO buffers, filter FIFO buffers, filter delay lines
36 kbit Block RAM	18000	488	Multi-rate FIFO

Memory Organization

In addition to distributed RAM, the Virtex™-5 device has a large number of 36kB block RAMs, each of which contain two independently controlled, 18 kB RAMs. The total memory configuration is given in Table 5.8. The 18 kB RAMs have been implemented in such a way, that the blocks can be configured to act as one 36 kB block RAM without the use of programmable interconnect. Block RAMs are placed in columns and can be cascaded to create deeper and wider RAM blocks. Each 18 kB block RAM, dual-port memory consists of an 18 kB storage area and two completely independent access ports along with other circuitry to allow the full expected RAM functionality to be achieved (See Figure 5.21). The full definition in terms of access pins is given below, and represents a standard RAM configuration.

A *clock* for each 18 kB block RAM which can be configured to have rising or falling edge. All input and output ports are referenced to the clock.
An *enable* signal to control the read, write, and set/reset functionality of the port with an inactive enable pin, implying that the memory keeps the previous state.
An additional enable signal called the *byte-wide write enable* signal which controls the writing and reading of the RAM in conjunction with the *enable* signal
The *register enable pin* which controls the optional output register.
The *set/reset* pin which forces the data output latches to contain a set value.
The *address bus* which selects the memory cells for read or write; its data bit width is decided by the size of RAM function chosen.

In latch mode, the read address is registered on the read port, and the stored data is loaded into the output latches after the RAM access time. When using the output register, the read operation will take one extra latency cycle. The write operation is also a single clock-edge operation with the write address being registered on the write port, and the data input is stored in memory. The additional circuitry highlighted in Figure 5.21, shows how inverted clock can be supported along with a registered output. The contents of the RAM can be initialized using the *INIT* parameter and can be indicated from the HDL source code.

The RAM provides a number of options for RAM configuration, some of which are listed in Table 5.9; the table shows how bit data width is traded off for memory depth, i.e. number of memory locations.

Dedicated logic has also been included in the block RAM enables, to allow the creation of synchronous or asynchronous FIFOs; these are important in some high-level design approaches, as will be seen later. This dedicated logic avoids use of the slower programmable CLB logic and routing resource, and generates the necessary hardware for the pointer write and read generation

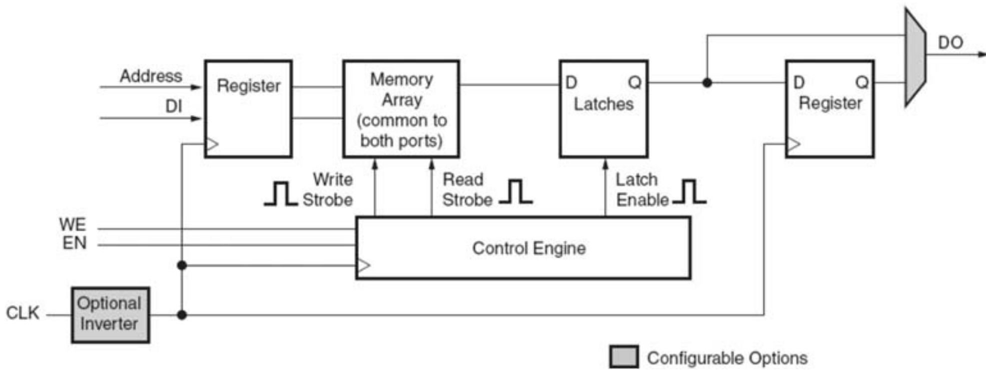


Figure 5.21 Block RAM logic diagram(Xilinx Inc. 2007b). Reproduced by permission of Xilinx Inc.

Table 5.9 Memory sizes for Xilinx Virtex™-5 block RAM

Data width	Memory depth
1 (cascade)	32768 (65 536)
2	16384
4	8192
9	4096
18	2048
36	1024
72	512

along with the setting of the various flags associated with FIFOs. A number of FIFO sizes can be inferred, including 8KX4, 4KX4, 4KX9, 2KX9, 2KX18, 1KX18, 1KX36, 512X36 and 512X72.

Virtex™-5 DSP Processing Resource

In addition to the scalable adders in the CLBs, the Virtex™-5 also provides a dedicated DSP processing block called DSP48E. The Virtex™-5 can have up to 640 DSP48E slices which are located at various positions in the FPGA, and supports many independent functions including multiply, MAC, multiply add, three-input add, barrel shifting, wide-bus multiplexing, magnitude comparator, bit-wise logic functions, pattern detect, and wide counter. The architecture also allows the multiple DSP48E slices to be connected together to form a wider range of DSP functions, such as DSP filters, correlators and frequency domain functions.

A simplified version of the DSP48E processing block is given in Figure 5.22. The basic architecture of the DSP48E block is a multiply–accumulate core, which is a very useful engine for many DSP computations. However, in addition to the basic MAC function, the DSP48E block also allows a number of other modes of operation, as summarized below:

- 25-bit x 18-bit multiplication which can be pipelined
- 96-bit accumulation or addition or subtracters (across two DSP48E slices)

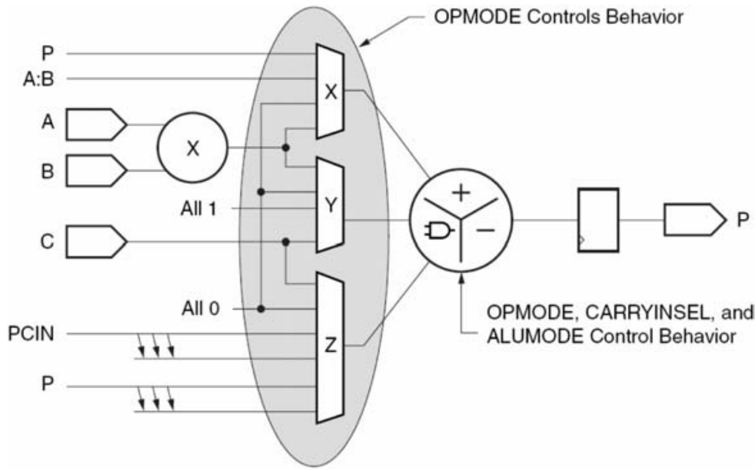


Figure 5.22 DSP processing blocks called DSP48E(Xilinx Inc. 2007c). Reproduced by permission of Xilinx Inc.

- triple and limited quad addition/subtraction
- dedicated bitwise logic operations
- arithmetic support for overflow/underflow

Each DSP48E slice has a 25-bit X18-bit multiplier which is fed from two multiplexers; the multiplexers accept a 30-bit A input and a 18-bit B input either from the switching matrix or from the DSP48E directly below. These can be stored in registers (not shown in Figure 5.22) before being fed to the multiplier. Just before multiplication, the A signal is split and only 25 bits of the signal are fed to the multiplier. A fast multiplier technique is employed which produces an equivalent 43-bit two's complement result in the form of two partial products, which are then sign-extended to 48 bits in the X multiplexer and Y multiplexer respectively before being fed into three input adder/subtracter for final summation.

As illustrated in Chapter 4, many fast multipliers work on the concept of using fast carry-save adders to eventually produce a final *sum* and *carry* signals, and then using a fast carry ripple to perform the final addition. This final addition is costly, either in terms of speed or if a speed-up technique is employed, then area. By postponing the addition to the ALU stage, a two-stage addition can then be avoided for multiply–accumulation, by performing a three-stage addition to compute the final multiplication output and an addition for the accumulation input *in one stage*. Once again, for flexibility, the adder/subtracter unit has been extended to function as a *arithmetic logic unit* (ALU), thereby providing more functionality at little hardware overhead. As the final stage of the conventional multiplication is being performed in the second-stage adder, a three-input addition is required with the third input used to complete the MAC operation if required.

The multiplexers allow a number of additional levels of flexibility to be added. For example, the P input can be used to feed in an input either from another DSP48E block from below using the PCIN in the Z multiplexer or looped back from the current DSP48E block say, for example, if a recursion is being performed using the P input to the Z multiplexer. The multiplier can be bypassed if not required, by using the A:B input which is a concatenation of the two input signals A and B, 25-bit and 18-bit words respectively; this gives a 43-bit word size which is the same as the multiplier output. Provision to initialize the inputs to the ALU to all 0s or all 1s,

is also provided. To increase the flexibility of the unit, the adder can also be split into several smaller adders, allowing two 24-bit additions or four 12-bit additions to be performed. This is known as the *SIMD mode*, as a single operation namely addition, is performed on multiple data, thus giving the *SIMD* operation. The DSP48E slice also provides a right-wire-shift by 17, allowing the partial product from one DSP48E slice to be shifted to the right and added to the next partial product, computed in an adjacent DSP48E slice. This functionality is useful, when the dedicated multipliers are used as building blocks, in constructing larger multipliers.

The diagram in Figure 5.22 is only basic, and does not indicate that other signals are also provided, in addition to the multiply or multiply–accumulate output, *P*. These include:

The cascadable A data port called *ACOUT*, which allows the A internal value to be fed directly to the output. Given that the A signal has been internally delayed, this would provide the delay chain for DSP functions e.g. a FIR filter.

Cascadable carryout (*CARRYCASCOUT*) and sign (*MULTSIGNOUT*) signals which are internal signals used to indicate the carryout and sign, when supporting 96-bit addition/subtraction across two DSP48E slices.

Up to four carry out signals (*CARRYOUT*) to support the *SIMD mode* of addition where up to four separate adders will need to generate carry out signals.

A pattern detector provides support for a number of numerical convergent rounding, overflow/underflow, block floating-point, and support for accumulator terminal count (counter auto reset) with pattern detector outputs (*PATTERNDETECT* and *PATTERNBDETECT*), to indicate if a pattern has been met and separate signals for overflow (*OVERFLOW*) and underflow (*UNDERFLOW*).

From a functional perspective, the synthesis tools will largely hide the detail of how the design functionality is mapped to the FPGA hardware, but it is important to understand that the level of functionality that is available as it determines the design approach the user will adopt. A number of detailed examples are listed in the relevant user guide (Xilinx Inc. 2007c), indicating how performance can be achieved.

Clock Networks and PLLs

The Xilinx Virtex™-5 FPGA family can provide a clock frequency of 550 MHz. The clock domains in the Virtex™-5 FPGA are organized into *six clock management* tiles or CMTs, each of which contain two *digital clock managers* (DCMs) and one PLL. In total, the FPGA has eighteen total clock generators.

A key feature of the Xilinx Virtex™-5 FPGA is the DCM, which provides a wide range of powerful clock management features including a *delay-locked loop* (DLL); this acts to align the incoming clock to the produced clock as described earlier. It also allows a range of clock frequencies to be produced, including a doubled frequency a range of fractional clock frequencies specifically 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, 6, 6.5, 7, 7.5, 8, 9, 10, 11, 12, 13, 14, 15, or 16 of the input clock. Coarse (90°, 180° and 270°) fine-grained phase shifting and various types of fine-grained or fractional phase-shifting are supported.

The PLL's main purpose is to act as a frequency synthesizer and to remove jitter from either external or internal clocks, in conjunction with the DCMs. With regard to clock generation, the six PLL output counters are multiplexed into a single clock signal for use as a reference clock to the DCMs. Two output clocks from the PLL can drive the DCMs; for example, one could drive the first DCM while the other could drive the second DCM. Flexibility is provided to allow the output of each DCM output to be multiplexed into a single clock signal, for use as a reference clock to the

PLL, but one DCM can be used as the reference clock to the PLL at any given time (Xilinx Inc. 2007c).

I/O and External Memory Interfaces

As with the Altera FPGA device, the Virtex™-5 FPGA supports a number of different I/O standard interfaces termed *SelectIO™ drivers* and *receivers*, allowing control of the output strength and slew rate and on-chip termination. As with the Altera FPGA, the I/Os are organized into a bank comprising 40 IOBs which covers a physical area that is 20 CLBs high, and is controlled by a single clock. The Virtex™-5 FPGA also includes *digitally controlled impedance* (DCI) technology, allowing the output impedance or input termination to be adjusted, and therefore, accurately match the characteristic impedance of the transmission line. The need to effectively terminate PCB trace signals, is becoming an increasing important issue in high-speed circuit implementation, and this approach purports to avoid the need to add termination resistors on the board. A number of standards are supported, including *low-voltage transistor–transistor logic* (LVTTL), *low-voltage complementary metal oxide semiconductor* (LVCMOS), *peripheral component interface* (PCI) including PCIX, PCI33, PCI66, and *low-voltage differential signalling* (LVDS), to name but a few.

Input serial-to-parallel converters (ISERDES) and *output parallel-to-serial converters* (OSERDES) are also supported. These allow very fast external I/O data rates such as SDR and DDR, to be fed into the internal FPGA logic which may be running an order of magnitude slower. This is essentially a serial-to-parallel converter with some additional hardware modules that allow reordering of the sequence of the parallel data stream going into the FPGA fabric, and circuitry to handle the strobe-to-FPGA clock domain crossover.

5.5 Lattice® FPGA Families

Lattice® offer a number of FPGA architectures, including the ispXPLD™ 5000MX family which extends the CPLD concept, the LatticeSC/M family which is a more of a standard FPGA with additional high-speed communications, memory and dedicated ASIC block implementation, and the Lattice ECP2/M family which is a low cost FPGA with a number of the features outlined in the LatticeSC/M family.

5.5.1 Lattice® ispXPLD 5000MX Family

The ispXPLD™ 5000MX family offers a combination of E^2PROM nonvolatile cells which stores the device configuration, and SRAM technology to provide the logic implementation, giving a solution that provides logic availability at boot-up. It also includes flexible memory capability, supporting single- or dual-port SRAM, FIFO, and ternary content addressable memory (CAM) operation along with dedicated arithmetic functionality. However, the technology benefits from the main attraction of using the CPLD architecture to provide predictable deterministic timing. The architecture of ispXPLD 5000MX device (Figure 5.23), consists of units called *multi-function blocks* (MFBs) interconnected with a *global routing pool* which are connected via multi sharing arrays (MSAs) to the input and output pins. The MFB consists of a multi-function array and associated routing, which can cope with up to 68 inputs from the GRP and the four global clock and reset signals and produce outputs to the macrocells or elsewhere. The device allows cascading of adjacent MFBs to support wider operation. Each MFB can be configured in a number of modes, including logic and memory configurations e.g., single- and dual-port RAM, FIFO Mode and CAM.

This description concentrates on the LatticeSC/M FPGA (Lattice Semi. Inc. 2007), as it represents the high-performance FPGA family, with many of the features which appear on the lower cost

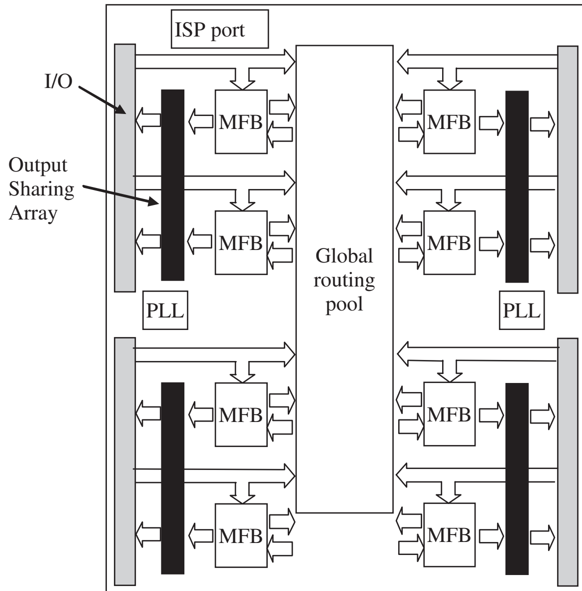


Figure 5.23 Lattice ispXPLD 5000MX family

Lattice ECP2/M families. The architecture of LatticeSC/M device is given in Figure 5.24. It comprises the standard programmable I/O block which is connected to rows of logic blocks organized as *programmable functional units* or PFUs. These PFUs comprise slices, each of which comprise two 4-input LUTs and two registers, along with carry propagate and carry generate signals which allow the creation of fast adder structures. As with other FPGA offerings, this functionality can be used to create combinational and sequential logic with the capability to scale LUT table sizes and register dimensions as required. As in the ispXPLD™ 5000MX family, the PFUs can also be configured to act as memory types. The largest device offers 115 k of LUTs.

In addition to the programmable logic, the device also incorporates up to 7.8 Mb of embedded block RAM, to match the 2 Mb of distributed RAM contained within the PFUs. These sysMEM EBRs as they are called, can be configured as RAM, ROM or FIFO, allowing a high level of programmability. In addition to the standard programmable I/O pins, high performance I/O is included in the form of dedicated SERDES and PCS hardware which provides 2Gbps I/O capability. This matches the concept of gathering large dedicated IP functionality on IP cores.

A feature that is different from offerings from Altera and Xilinx, is the structured ASIC capability as highlighted in Figure 5.25. This *masked array for cost optimization* or MACO block is a sea of 50 000 ASIC gates which has been created using a 90 nm CMOS process technology and optimized for speed, power dissipation, and area (Lattice Semi. Inc. 2006). The MACO block interfaces directly to the I/O and also to the FPGA fabric, thereby allowing dedicated fast low-power implementation for specialized hardware. In addition to the gates, each MACO block contains three 64×40 asynchronous dual-port RAMs in addition to the RAM of the FPGA; the dual-port RAMs are co-located to the MACO block and can be accessed through the dedicated *MACO interface block* or MIB. By being located between the I/O pins and the on-board block RAM makes this ideal for implementing a number of fast, lower-power blocks such as dedicated or specialized memory interfaces.

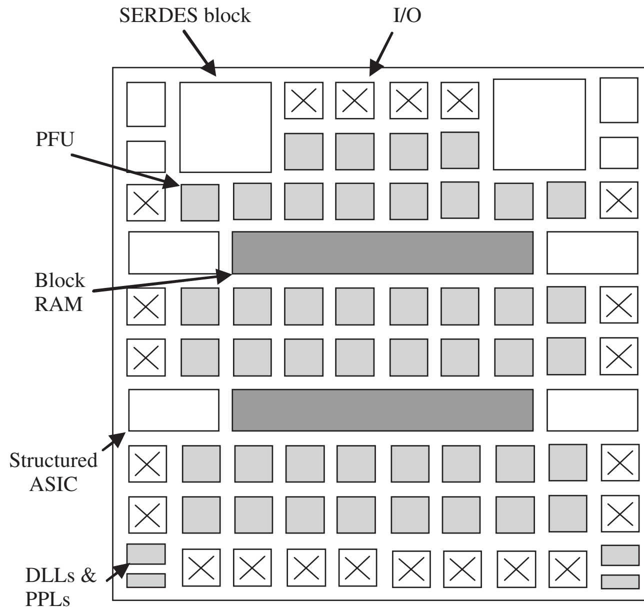


Figure 5.24 LatticeSC/M Family

5.6 Actel® FPGA Technologies

Actel® offers a number of FPGA technologies, based on flash and antifuse technologies. They have also recently launched a Fusion™ technology which represents the first *mixed signal* FPGA (Actel Corp. 2007a). It comprises A/D converters, embedded flash memory and as well as more conventional digital FPGA hardware in the form of D-type flip-flops and RAM. The flash technology is nonvolatile, meaning that the FPGA stores its design and is live at power-up, without the need to be programmed from a ROM or co-processor. The largest device has 1.5 M system gates with 270 kbits of dual-port SRAM, up to 8 Mbits of flash memory, 1 kbit of user flash ROM, and up to 278 user I/Os.

5.6.1 Actel® ProASIC^{PLUS} FPGA Technology

The ProASIC^{PLUS} FPGA technology is based on flash technology. It is similar to E^{2}PROM technology in that it stores its charge and therefore allows the device to store its program. The FPGA technology is organized with an architecture comparable to gate arrays, as illustrated in Figure 5.25, and comprises a sea of tiles where each tile can be configured as a three-input logic function, or a D-type flip-flop. As the diagram shows, the architecture comprises a grid of tiles with a number of embedded two-port SRAM blocks, top and bottom which allow synchronous and asynchronous operation. The tiles consist of a number of multiplexers and logic gates, sufficient to allow the creation of a flip-flop with the necessary globally connected reset and clock signalling. The tiles connect to both the local and the longer line routing.

The tiles are connected by a hierarchy of routing that has four levels which are organized in terms of length; it would appear to have a similar organisation to the older Xilinx XC62000 FPGA technology which was a similarly fine-grained architecture. The next level of lines run one, two or

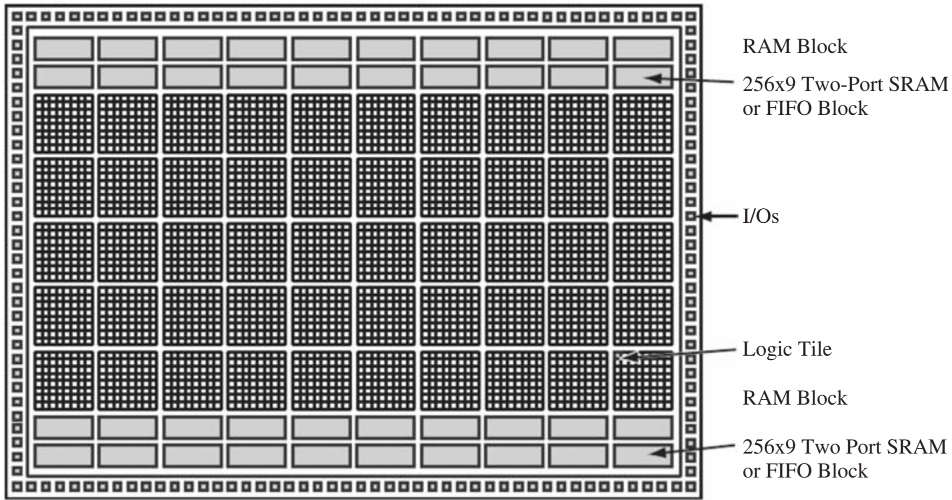


Figure 5.25 Actel® ProASIC^{PLUS} FPGA (Actel Corp. 2007b). Reproduced by permission of Actel Corp.

four tiles. Both of these are accessible at the tile output. The next level of interconnect are long-line resources which span the length of the chip. The final level is global signals, which by definition have to be low-delay, as they will be used for global signals such as clock, reset and enable signals which would be used globally and would degrade the performance of the chip.

The device also incorporates two clock conditioning blocks which, like other FPGA devices, comprise PLLs and delay circuits for synchronizing the external clock as well as multiplier/dividers circuits along with necessary circuitry to connect to the global routing network. The device comes with four clock networks or global trees, specifically designed for distributing a low-latency, low-skew clock signal.

A key aspect of the implementation process with such a fine-grained technology, is the good placement of the design units into the hardware, as the limited routing organization can hamper design quality and utilization of the underlying hardware blocks. For this reason, the software allows the use of constraints to control the placement of the design. The concept of generating a circuit architecture to best match the FPGA resource requirements, is definitely needed, not only to achieve the necessary performance, but also to alleviate the software effort in achieving an efficient implementation. As with other FPGAs, a range of I/O blocks is available and boundary scan in the form of JTAG is provided for system text at the board level.

5.6.2 Actel® Antifuse SX FPGA Technology

In addition to flash memory devices, Actel also provide an antifuse technology which is once-only programmable. An antifuse is a two-terminal device with an unprogrammed state presenting a very high resistance between its terminals. Typically as shown in Figure 5.26, two conductors are separated by mostly insulator, but at certain points by amorphous or 'programmable' silicon. When a high voltage is applied across its terminals, the antifuse will 'blow' and create a low-resistance link (as opposed to an open circuit as in a fuse).

In the case of the Actel antifuse link shown in Figure 5.27, the connection is made up from a combination of amorphous silicon and dielectric material and has an 'on' state resistance of

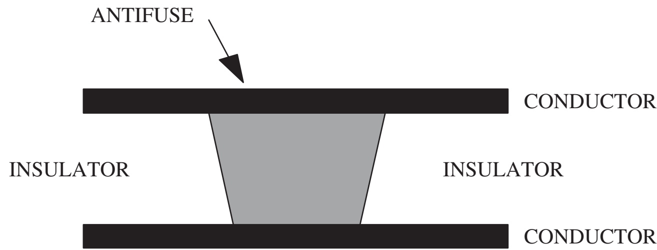


Figure 5.26 Basic antifuse link

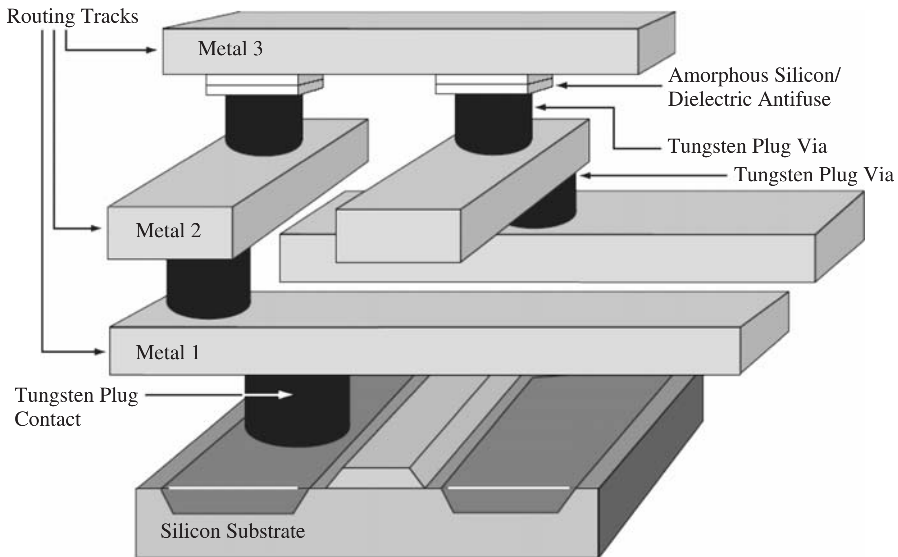
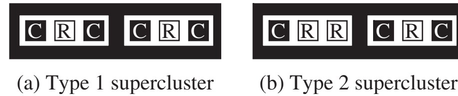


Figure 5.27 Actel® ProASIC^{PLUS} FPGA(Actel Corp. 2007c). Reproduced by permission of Actel Corp.

25Ω with a capacitance of 1.0 fF (Actel Corp. 2007c). The use of three-layer metal as shown in Figure 5.27 and the use of metal-to-metal antifuse results in better performance and smaller area. The metal-to-metal antifuse lowers the programmed resistances, thereby providing better speed, and the multiple metal layers now allows the placement of the antifuses above the logic, thereby avoiding routing channels. This provides better density of logic and lead to smaller devices with better, performance.

The architecture comprises a sea of modules which like the ProASIC^{PLUS} FPGA technology, is fine-grained with two types of cells, namely a C-cell which is effectively a combinational cell comprising a series of multiplexers and logic gates, and an R-cell which is a register cell which comprises a D-type flip-flop with various multiplexing hardware, to allow various connections to the input, and to allow various clock signals to be used to clock the cell. Because of the programmability to configure the cell, the fabric is comprised of a variety of the different cell types, meaning that pre-defined mappings of the cells must be created, called clusters. The two types of superclusters



(a) Type 1 supercluster (b) Type 2 supercluster

Figure 5.28 Cluster organization

are illustrated in Figure 5.28; the one supercluster comprises two C, R and C clusters and the other supercluster comprises a C, R and R cluster followed by a C, R and C cluster.

In this structure, a number of different level of routing is given. The first type of routing comprises the *DirectConnect* which is a horizontal routing resource that connects from C-cell to its neighbouring R-cell in a cluster, and the *FastConnect* routing which provides vertical routing. Two globally orientated routing resources are also provided, known as *segment routing* and *high-drive routing*, which as the names suggest are for smaller routes, i.e. *segments* and more global routes. As with the ProASIC*PLUS* FPGA technology, consideration has to be given to the careful placement of the cells in order to achieve efficient implementation, otherwise the disadvantage of the delays of programmable routes will have to be suffered.

Clock rates of 300 MHz are quoted in (Actel Corp. 2007c). In addition, it is also argued that security is a key feature of the technology, as it proves difficult to *reverse engineer* the device because it is hard to distinguish between the programmed and unprogrammed antifuses, and there is no configuration bitstream to intercept.

5.7 Atmel® FPGA Technologies

Atmel offer a range of FPGA technologies, ranging from the AT40K and AT40KAL series co-processor FPGAs range which offer the concept of what is termed FreeRAM™ which can be used without infringing the available logic resource. Their FPGA technology can be used as an embedded core in the form of FPSLIC™ FPGA family, which provides from 5 k up to 50 k gates, up to 36 k of SRAM and a 25 MHz AVR MCU. The AT6000 series FPGAs are marketed as reconfigurable DSP co-processors, as they offer register counts of 1024 to 6400 registers, making them ideal for use as computing DSP functions that have been off-loaded into hardware. One of the key features of the AT6000, AT40K and AT40KAL FPGA families is that they offer reconfigurability, allowing part of the FPGA to be reprogrammed without loss of register data, whilst the remainder of the FPGA continues to operate without disruption. For this reason, the AT40K FPGA family are considered in a little more detail.

5.7.1 Atmel® AT40K FPGA Technologies

The AT40KAL is a SRAM-based FPGAs with distributed dual-port/single-port SRAM and eight global clocks with only one global reset. The family ranges in size from 5000 to 50 000 usable gates. The AT40KAL is a fine-grained FPGA architecture comprising simple cells organized into 4×4 grids, each of which are surrounded by repeater cells, as shown in Figure 5.29 (Atmel Corp. 2006). The repeaters regenerate the signals and allow connection of any bus to any other bus, on the same plane. Each repeater has connections to two adjacent local-bus segments, which provide localized connections in the four cells shown, and two express-bus segments for longer line connection which spans eight cells.

The core cell is very fine-grained, comprising two 3-input LUTs (8×1 ROM) which can be configured as a 4-input LUT, a D flip-flop, a 2-to-1 multiplexer and an AND gate for implementing

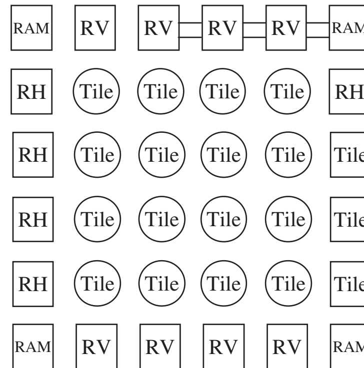


Figure 5.29 Atmel AT50K FPGA 4 × 4 cell

multiplier arrays. As with the other FPGA technologies, the combination of LUT and D-type flip-flops allow a wide range of combinational and sequential logic to be implemented. There is a DSP mode, but it is basic, compared with other technologies, as it only allows the generation of an array multiplier using the cells, resulting in a relatively poor DSP performance compared with other FPGA technologies.

At the intersection of each repeater row and column, there is a 32 × 4 RAM block accessible by adjacent buses, which can be individually addressed through the provision of a series of local, and express horizontal and vertical buses. Reading and writing of the dual-port FreeRAM are independent of each other, and reading is completely asynchronous.

An interesting aspect of the Atmel® AT40K FPGA technology is the fact that it can be partially reconfigured, i.e. programmed, allowing the function of the design or part of the design to be changed whilst it is operating. This is worth examining in a little more detail.

5.7.2 Reconfiguration of the Atmel® AT40K FPGA Technologies

The AT40K FPGA technology has four basic configuration modes of operation (Atmel Corp. 2006). First, there is *power-on* reset when the device is first powered up which involves a complete reset of all of the internal configuration SRAM. Second, the same reset sequence can be invoked by the manual reset, via the reset pin. The third configuration mode is configuration download, where the FPGA's configuration SRAM is programmed using serial, or parallel data via its input pins. The fourth mode is when no configuration is active. The AT40K FPGA allows complete reconfigurability down to the byte level.

The CacheLogic® architecture lets users reconfigure part of the FPGA, while the rest of the FPGA continues to operate unaffected; this done using a *windowing* mechanism. This allows the user to load the SRAM memory map in smaller segments, allowing overwriting of portions of the configuration SRAM that is not being used, with new design information. In synchronous RAM mode, the device receives a 32- or 40-bit-wide bitstream composed of a 24-bit address and either an 8-bit-wide or 16-bit-wide dataword. Address, data and write enable are applied simultaneously at the rising edge of CCLK. In this mode, designed to interface to a generic IO port of a microprocessor, the FPGA configuration SRAM is seen as a simple memory-mapped address space. The user has full read and write access to the entire FPGA configuration SRAM. The overhead normally associated with bitstreams is eliminated, resulting in faster reconfiguration.

5.8 General Thoughts on FPGA Technologies

The chapter has covered a number of FPGA technologies from a variety of companies, but highlighting the two major vendors, namely Xilinx and Altera. The major drive in FPGA technology, has been a move from an FPGA being a fine-grained device where simple logic was implemented using LUTs and programmable interconnect, to one where it is a collection of heterogeneous complex units such as dedicated DSP blocks, high-speed communications blocks, soft and hard processor and of course, the previously mentioned LUTs.

This has had a number of implication for the design process. With earlier FPGAs, the target has been to develop an efficient implementation where utilization of FPGA hardware had been the main focus. This involved using, initially, schematic design capture packages and then more recently HDL-based tools, to achieve an efficient design implementation. Given that the focus was to achieve high utilization, considerable effort was expended to utilize the underlying LUTs. Thus, a number of circuit-level design techniques were developed to best use LUTs, and flip-flops for that matter, to achieve the implementation. These techniques are covered in more detail in Chapter 6 as it is important to understand the principles, even though a lot of this activity is carried in synthesis tools. For this reason, the design techniques are only briefly covered.

As complexity has grown with more recent FPGAs, the major challenge has been to create a circuit architectures at one level (see Chapter 8) and then system-level architectures at another level which is a major focus in Chapters 7, 9 and 11. There has been a major focus on generating circuit architectures that contain all of the FPGA specific details from a SFG representation of the DSP algorithm. Since the major performance gains of FPGAs in the area of DSP implementation, this involves exploiting concurrency in terms of parallelism and pipelining. The processes for achieving this are described in detail in Chapter 8; the higher-level tools flows are then described in the later chapters.

References

Actel Corp. (2007a) Fusion family of mixed-signal flash FPGAs with optional soft arm support. Web publication downloadable from www.actel.com.

Actel Corp. (2007b) Proasicplus flash family FPGAs. Web publication downloadable from www.actel.com.

Actel Corp. (2007c) Sx family FPGAs. Web publication downloadable from www.actel.com.

Altera Corp. (2000) Max ii device family data sheet. Web publication downloadable from http://www.altera.com.

Altera Corp. (2007) Stratix iii device handbook. Web publication downloadable from http://www.altera.com.

Atmel Corp. (2006) 5k–50k gates coprocessor FPGA with freeram. Web publication downloadable from http://www.atmel.com.

Bouvier D (2007) Rapidio: the interconnect architecture for high performance embedded systems. Web publication downloadable from http://www.techonline.com/.

Lattice Semi. Inc. (2006) Delivering FPGA-based pre-engineered IP using structured ASIC technology. Web publication downloadable from www.latticesemi.com.

Lattice Semi. Inc. (2007) Latticesc/m family data sheet. Web publication downloadable from www.latticesemi.com.

Rose J, Francis RJ, Lewis D and Chow P (1990) Architecture of field programmable gate arrays: the effect of logic block functionality on area efficiency. *IEEE Journal of Solid State Circuits* **25**(5), 1217–1225.

Xilinx Inc. (2007a) Achieving higher system performance with the virtex-5 family of FPGAs. Web publication downloadable from www.xilinx.com/bvdocs/whitepapers/wp245.pdf.

Xilinx Inc. (2007b) Virtex-5 user guide. Web publication downloadable from www.xilinx.com.

Xilinx Inc. (2007c) Xilinx ug193 virtex-5 xtremedsp design considerations: User guide. Web publication downloadable from www.xilinx.com/bvdocs/userguides/ug193.pdf.

6

Detailed FPGA Implementation Issues

6.1 Introduction

The previous chapters have set the scene in terms of background to DSP and computer arithmetic, and then in the last two chapters, the various implementation technologies have been highlighted; Chapter 4 has highlighted the wider range of technologies and Chapter 5 has described, in a little more detail, the various FPGA offerings. The remaining chapters now describe the issues for implementing complex DSP systems onto heterogeneous platforms, or even a single FPGA device. This encompasses considerations such as selection of the suitable model for DSP system implementation, partitioning of DSP complexity into hardware and software, mapping of DSP functions efficiently onto FPGA hardware, development of a suitable memory architecture, and achievement of design targets in terms of throughput, area and energy. However, it is imperative that the reader understands the detailed FPGA implementation of DSP functionality in order that this process is inferred correctly at both the system partitioning and circuit architecture development stages.

At the system partitioning level, for example, it may become clear that the current system under consideration will consume more than the dedicated multiplicative resources available. The designer is then faced with a number of options, either to restrict the design space so that the design mapping ensures that only the dedicated multiplier resources are used, or alternatively, to map the design to the existing FPGA resources, in order to create the additional multipliers using LUTs and dedicated adders. Moreover, it may be that the additional multiplicative operations are fixed in nature, which means that constant coefficient multipliers could be used which need considerably less LUTs.

From a circuit architecture perspective, the key objective is to achieve faster implementation, where area against speed is a key trade-off. It is essential to be aware of the optimizations available in producing the required design as this could determine the trade-off between implementing the design in FPGA hardware or in software, which may be a critical decision for the system. It is clear from published work that the decision is not obvious unless a lot of detailed work is undertaken to investigate the FPGA implementation; however, this is clearly not practical at system level as the design detail is severely limited. Indeed, it was the efficient mapping in the earlier LUT-based devices such as the Xilinx XC4000 that first led to the concept of achieving performance gains over existing DSP processor devices (Goslin 1995); this was demonstrated for some DSP functions such as FIR filters. With modern day synthesis tools, a lot of these optimizations are contained within the synthesis tools, but it is important that the underlying principles are understood.

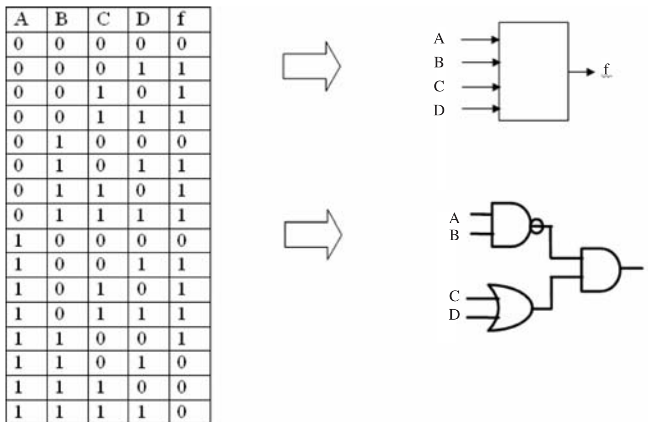
FPGA-based Implementation of Signal Processing Systems R. Woods, J. McAllister, G. Lightbody and Y. Yi
© 2008 John Wiley & Sons, Ltd

Finally, the utilization of the design may be an issue: indeed it could be argued that, for a cost-effective design, it should always be so, otherwise the designer is selecting a larger FPGA device than needed, thereby increasing cost! In this case, optimization to achieve the last few nanoseconds of timing from the design may be the critical aspect. Of course, a lot of these optimizations are now increasingly buried within synthesis tools.

A trivial example is the design of a *walking one* circuit. In a typical design environment, one would be tempted to describe this design as a finite state machine (FSM) where each state will have one bit high. The synthesis tool will treat this as FSM machine with one hot encoding, but it is clear that the design can be achieved by writing the code for a register chain where one of the flip-flops is set logic '1' using preset, with others being reset. This comes from the understanding that the FPGA flip-flop has a set–preset facility. The synthesis tool might be able to come with an efficient design, but it is clear that any doubt that the synthesis tool would give this efficient solution can be removed by coding critical parts of the circuit in this direct way. Of course, this could be viewed as working against the design principle of writing the code to be as portable as possible. Therefore, the focus of this chapter is to introduce and cover in reasonable detail some of these technology-specific design optimizations and show how the functionality is mapped into FPGA fabric. In particular, the chapter will look at the implementation of memory in FPGAs as this tends to be a key issue in many applications. In addition, the use of advanced design techniques for DSP functions such as *distributed arithmetic* (DA) and *reconfigurable mux* technique for building coefficient specific DSP functions, are also covered.

6.2 Various Forms of the LUT

The reason why most FPGA vendors choose the LUT as the logic fundamental building block, is that an n-input LUT can implement any n-input logic function. As shown in Figure 6.1, the use of LUTs as compared with logic gates, places different constraints on the design. For example, the diagram shows how the function can be mapped into logic gates, where number of gates is the design constraint. This is irrelevant from a LUT implementation perspective however, as the underlying criteria in determining the number of LUTs is directly related to number of inputs and, in some cases, the number of outputs in the design rather than the logic complexity. For the early Xilinx devices, a 4-input LUT with the capability to extend this efficiently to a 5-input LUT was



A	B	C	D	f
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Figure 6.1 Mapping logic functions into LUTs

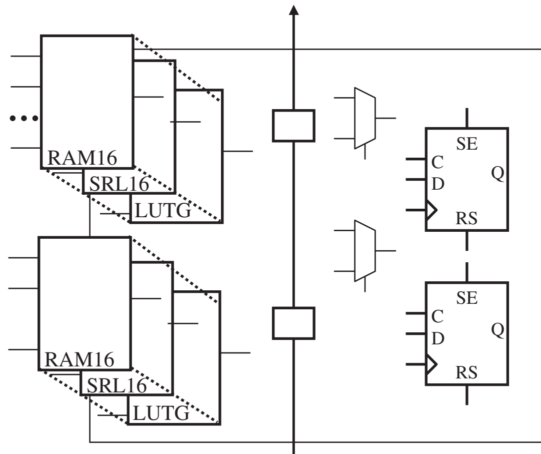


Figure 6.2 Additional usage of CLB LUT resources

the core unit, but recently, this has been extended to a 6-input LUT in the latest Xilinx Virtex™-5 FPGA technologies, and to an 8-input LUT in the latest Altera Stratix® III FPGA devices.

This represents however, only one of the reasons for using a n-input LUT as the main logic building block of FPGAs. Figure 6.2 illustrates the resources of the CLB for the family of Xilinx FPGA devices; this comprises two LUTs, shown on the left, two flip-flops, shown on the right and the fast carry adder chain, shown in the middle. The figure highlights how the LUT resource shown on the left-hand side, can be used as a LUT, but also as a shift register for performing a programmable shift and as a RAM storage cell.

The basic principle of how the LUT can be used as a shift register is covered in detail in the Xilinx application note (Xilinx Inc. 2005). First, the LUT can be thought of as 16:1 multiplexer as shown in Figure 6.3 where the address input (here a 4-bit input as the note is dealing with a Xilinx Spartan device) is used to address the specific input stored in the RAM. In this case, the contents of the multiplexer would have been treated as being fixed. In the case of the SRL16, the Xilinx name for the 16-bit shift register, the fixed LUT values are configured instead as an addressable shift register, as shown in Figure 6.3(b).

The shift register inputs are the same as those for the synchronous RAM configuration of the LUT given in (Xilinx Inc. 2005) namely, a data input, clock and clock enable. The LUT uses a special output called Q15 in the Xilinx library primitive device which is in effect the output provided from the last flip-flop.

The design works as follows. By setting up an address, say 0111, the value of that memory location is read out as an output and at the same time, a new value is read in which is deemed to be the new input, *DIN* in Figure 6.3(b). If the next address is 0000 and the address value is incrementally increased, it will take 8 clock cycles until the next time that 0111 address corresponding to an shift delay of 8. In this way, the address size can mimic the shift register delay size. So rather than shift all the data as would happen in a shift register, the data is stored statically in a RAM and the changing address line mimics the shift register effect by reading the relevant data out at the correct time.

Details of the logic cell SRL structure are given in Figure 6.4 which refers to the Xilinx Spartan FPGA family with a 4-input LUT. The cell has an associated flip-flop and a multiplexer which make up the full cell. The flip-flop provides a write function synchronized with the clock, and the additional multiplexer allows a direct *DI* input, or if a large shift register is being implemented,

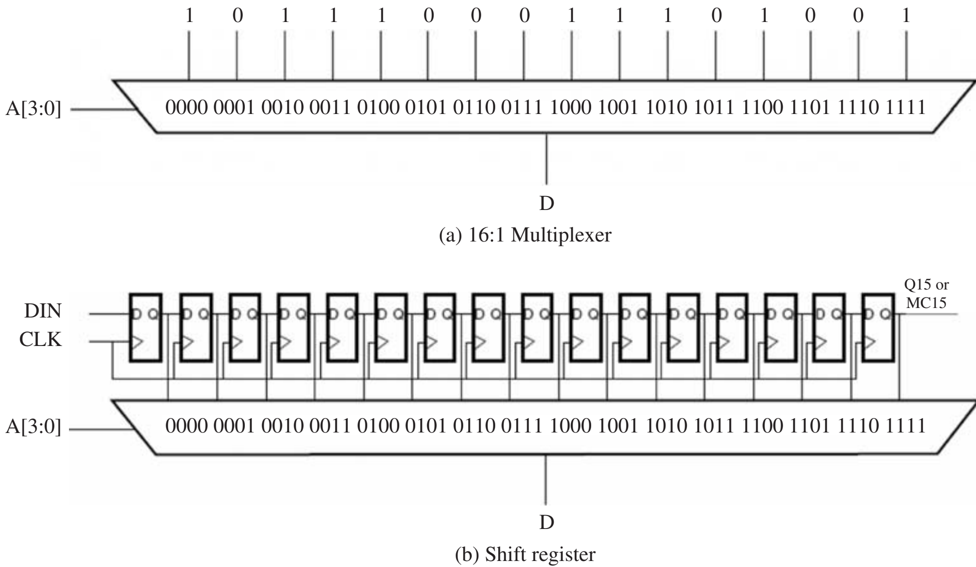


Figure 6.3 Configurations for Xilinx CLB LUT (Xilinx Inc. 2005). Reproduced by permission of Xilinx Inc.

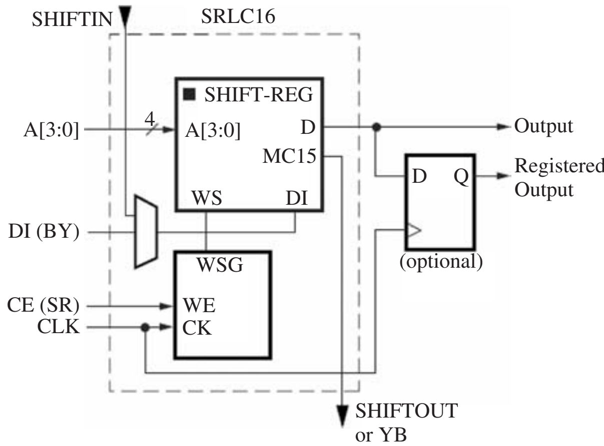


Figure 6.4 Detail SRL logic structure(Xilinx Inc. 2005). Reproduced by permission of Xilinx Inc.

an *SHIFTIN* input from the cell above. The address lines can be changed dynamically, but in a synchronous design implementation it would be envisaged that they would be synchronized to the clock.

This has huge implications for the implementation of DSP systems. As will be demonstrated in Chapter 8, the homogeneous nature of DSP operations is such that hardware sharing can be employed to reduce circuit area. In effect, this results in a scaling of the delays in the original circuit; if this transformation results in a huge memory increase, then the overall emphasis to

reduce complexity has been negated. Being able to use the LUTs as shift registers in addition to the existing flip-flops can result in a very efficient implementation. Thus, a key design criteria at the system level, is to balance the flip-flop and LUT resource in order to achieve the best utilization of the CLB usage. This can be seen in a number of designs later in the book.

6.3 Memory Availability

FPGAs offer a wide range of different types of memory, ranging from block RAM, through to highly distributed RAM in the form of the multiple LUTs available, right down to the storage of data in the flip-flops that are widely available in the FPGA fabric. As was demonstrated in the previous section, a trade-off can be performed between the LUT and flip-flop storage capability, but this will be for small distributed memories. There may be cases where there is a need to store a lot of input data, such as an image of block of data in image processing applications, or large sets of coefficient data, as in some DSP applications, particularly when multiplexing of operations has been employed. In these cases, the requirement is probably for large RAM blocks.

As illustrated in Table 6.1, FPGA families are now adopting quite large on-board RAMs. The table gives details of the DSP-flavoured FPGA devices, from both Altera and Xilinx. Considering both vendors' high-end families, the Xilinx Virtex™-5 and the Altera Stratix® FPGA technologies, it can be determined that block RAMs have grown from being a small proportion of the FPGA circuit area, to representing 1/15 or 1/10 of the circuit area. Typically these ratios tend to be lower for the logic-flavoured FPGA families. In addition to small distributed RAMs, the FPGA families also possess larger block RAM which have the advantage that they are dual-port, providing flexibility for some DSP applications. The Virtex-5 block RAM stores up to 36K bits of data and can be configured as either two independent 18 kb RAMs, or a 36 kb RAM. Each 36 kb block RAM can be configured as a $64\,k \times 1$ (when cascaded with an adjacent 36 kb block RAM), $32\,k \times 1$, $16\,k \times 2$, $8\,k \times 4$, $4\,k \times 9$, $2\,k \times 18$, or $1\,k \times 36$ memory. Each 18 kb block RAM can be configured as a $16\,k \times 1$, $8\,k \times 2$, $4\,k \times 4$, $2\,k \times 9$, or $1\,k \times 18$ memory.

This section has highlighted the range of memory capability in the two most common FPGA families. This provides a clear mechanism to develop a memory hierarchy to suit a wide range of DSP applications. In image processing applications, various sizes of memory are needed at different parts of the system. Take for example, the fast motion estimation circuit shown in Figure 6.5 where

Table 6.1 FPGA RAM size comparison for Xilinx Virtex 4 and 5 and Spartan FPGAs and Altera Stratix and Cyclone FPGAs

Family	Model	Block RAM (kb)	Distributed RAM (kb)	Mults	I/O	BRAM /LUT
Virtex	XC5VSX35T	3024	520	192	360	5.8
	XC5VSX95T	8784	1520	640	640	5.8
	XC4VSX25	2304	160	128	320	14.4
	XC4VSX55	5760	384	320	640	15.0
Spartan	XC3S100E	72	15	4	108	4.8
	XC3S1600E	648	231	36	376	2.8
Stratix	EP3SE50	5328	594	384	480	9.0
	EP3SE260	14688	3180	768	960	4.6
Cyclone	EP2C5	117	72	13	158	1.6
	EP2C70	1125	1069	150	622	1.1

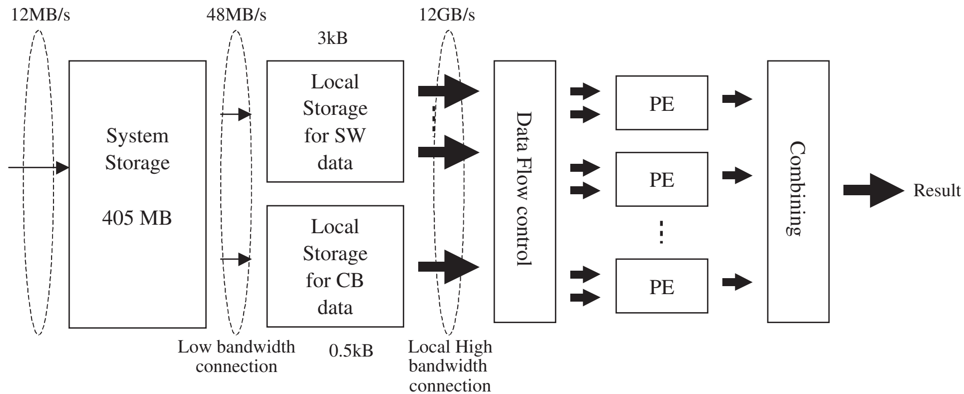


Figure 6.5 Dataflow for motion estimation IP core

the aim is to perform the highly complex motion estimation (ME) function on dedicated hardware. The figure shows the memory hierarchy and data bandwidth considerations for implementing such a system. In order to perform the ME functions, it is necessary to download the current block (CB) data and area where the matching is to be performed, namely the search window (SW), into local memory. Given that the SW is typically 24×24 pixels, the CB sizes are 8×8 pixels and a pixel is 8-bits, this corresponds to 3 k and 0.5 k memory files which would typically be stored in the embedded RAM blocks. This is because the embedded RAM is an efficient mechanism for storing such data and the bandwidth rates are not high.

In an FPGA implementation, it might then be necessary to implement a number of hardware blocks or IP cores to perform the ME operation, so this might require a number of computations to be performed in parallel. This requires smaller memory usage which could correspond to smaller distributed RAM or if needed, LUT-based memory and flip-flops. However, the issue is not just the data storage, but the data rates involved which can be high as illustrated in the figure. Smaller distributed RAM, LUT-based memory and flip-flops provide much high data rates as they each possess their own interfaces. Whilst this date rate may be comparable to the larger RAMs, the fact that each memory write or read can be done in parallel, results in a very high, data rate. Thus it is clear, that even in one specific application, there are clear requirements for different memory sizes and date rates; thus the availability of different memory types and sizes, such as those available in FPGAs, is vital.

6.4 Fixed Coefficient Design Techniques

Usually, a fully programmable multiplier capability is needed which is why FPGA vendors have now included quite of number of these on FPGAs. In some DSP applications however, there is sometimes the need to perform a multiplication of one word by a single coefficient value such as in a fixed filtering operation, or DSP transforms such as the DCT or FFT. In a processor implementation, this has little impact, but in dedicated hardware there is the chance to alter the hardware complexity needed to perform the task, thus dedicated coefficient multiplication or *fixed coefficient* multiplication (KCM) has the considerable potential to reduce the circuitry overhead. A number of mechanisms have been used to derive KCMs. These include: DA (Goslin and Newgard 1994), string encoding and common sub-expression elimination (Cocke 1970, Feher 1993).

This concept translated particularly well to earlier FPGA devices where only LUTs and dedicated fast adders were available for building multipliers; thus an area gain was achieved in implementing a range of these fixed coefficient functions (Goslin and Newgard 1994, Peled and Liu 1974). A number of techniques have evolved to these fixed coefficient multiplications and whilst a lot of FPGA architectures have dedicated multiplicative hardware on-board in the form of dedicated multipliers or DSP blocks, it is still worth briefly reviewing the approaches available. The section considers the use of DA which is used for single fixed coefficient multiplication (Peled and Liu 1974) and also the reduced coefficient multiplier (RCM) approach which can multiply a range of coefficient values (Turner and Woods 2004).

6.5 Distributed Arithmetic

Distributed arithmetic (DA) is an efficient technique for performing multiply-and-add in which the multiplication is re-organized such that multiplication and addition is performed on data and single bits of the coefficients, at the same time. The principle of the technique is based on the assumption that we will store the computed values rather than carry out the computation (as FPGAs have a readily supply of LUTs).

Assume that we are computing the sum of products computation in Equation (6.1) where the values x_j represent a data stream and the values a_0, a_1, … a_{N-1} represent a series of coefficient values. Rather than compute the partial products using AND gates, we can use LUTs to generate these and then use fast adders to compute the final multiplication. An example of a 8-bit LUT-based multiplier is given in Figure 6.6; it can seen that this multiplier would require a 4 kbits of LUT memory resource which would be considerable.

$$y = \sum_{i=0}^{N-1} a_i x_i \tag{6.1}$$

The memory requirement is vastly reduced (to 512 bits for the 8-bit example) when the coefficients are fixed which now makes this an attractive possibility for an FPGA architecture. The obvious implementation is to use a LUT-based multiplier circuit such as that in Figure 6.6 to perform the multiplication a_0x_0. However, a more efficient structure results by employing DA (which

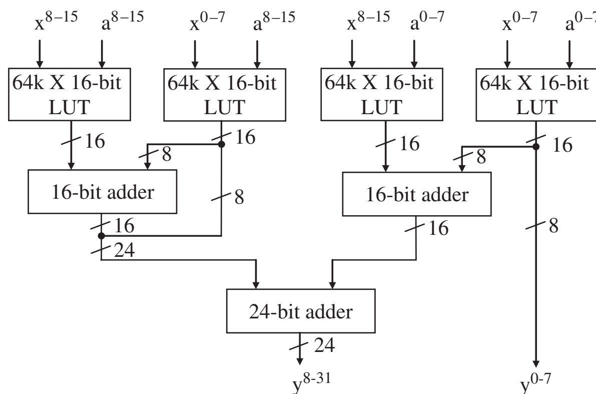


Figure 6.6 LUT-based 8-bit multiplier (Omondi 1994)

is described in detail in (Peled and Liu 1974, Meyer-Baese 2001, White 1989). The following analysis follows closely the approach described in White (1989). Again in the following analysis, remember that the coefficient values a_0, a_1, ... a_{N-1} represent a series of *fixed* coefficient values.

Assume the input stream x_n is represented by as a two's complement signed number which would be given as:

$$x_n = -x_n^0 + \sum_{j=1}^{M-1} x_n^j 2^j \quad (6.2)$$

where $x_n^j 2^j$ denotes the jth bit of x_n which is the nth sample of the stream of data x and x_n^0 denotes the sign bit and so is indicated as negative in the equation. The data wordlength is thus M bits. The computation of y can then be rewritten as:

$$y = \sum_{i=0}^{N-1} a_i \left(-x_n^0 + \sum_{j=1}^{M-1} x_i^j .2^j) \right) \quad (6.3)$$

Multiplying out the brackets, we get:

$$y = \sum_{i=0}^{N-1} a_i(-x_n^0) + \sum_{i=0}^{N-1} a_i \sum_{j=1}^{M-1} x_i^j 2^j \quad (6.4)$$

The fully expanded version of this is given below:

$$\begin{aligned} y = {} & a_0(-x_0^0) + a_0(x_0^1 2^1 + x_0^2 2^2 + \ldots + x_0^{M-1} 2^{M-1}) \\ & + a_1(-x_1^0) + a_1(x_1^1 2^1 + x_1^2 2^2 + \ldots + x_1^{M-1} 2^{M-1}) \\ & \vdots \\ & + a_{N-1}(-x_{N-1}^0) + a_{N-1}(x_{N-1}^1 2^1 + x_{N-1}^2 2^2 + \ldots + x_{N-1}^{M-1} 2^{M-1}). \end{aligned}$$

Reordering, we get:

$$y = \sum_{i=0}^{N-1} a_i(-x_i^0) + \sum_{j=1}^{M-1} \left[\sum_{i=0}^{N-1} a_i(x_i^j) \right] 2^j \quad (6.5)$$

and once again, the expanded version gives a clearer idea of how the computation has been reorganized as shown below:

$$\begin{aligned} y = {} & 2^0(a_0(-x_0^0) + a_1(-x_1^0) + \ldots + a_1(-x_{N-1}^0) \\ & + 2^{-1}(a_0 x_0^1 + a_1 x_1^1 + \ldots + a_{N-1} x_{N-1}^0) \\ & \vdots \\ & + 2^{-M+1}(a_0 x_0^{M-1} + a_1 x_1^{M-1} + \ldots + a_{N-1} x_{N-1}^{M-1}) \end{aligned}$$

Given that the coefficients are now fixed values, the term $\sum_{i=0}^{N-1} a_i(x_n^i)$ in Equation (6.5) has only 2^K possible values and the term $\sum_{i=0}^{N-1} a_i(-x_n^0)$ has only 2^K possible values. Thus the implementation can be stored in a LUT of size 2×2^K bits which for the earlier 8-bit example would represent 512 bits of storage.

Consider an implementation where $N = 4$ in order to see how it fits into the FPGA LUT-based architecture. This gives the expanded expressions for each of the terms in Equation (6.6) as follows:

$$\sum_{i=0}^{N-1} a_i(x_i^j) = a_0(x_0^0) + a_1(x_1^0) + a_2(x_2^0) + a_3(x_3^0) \tag{6.6}$$

This then shows that if we use the x inputs as the addresses to the LUT, then the stored values are those shown in Table 6.2. By rearranging Equation (6.7) to achieve the representation in Equation (6.8), we see that the contents of the LUT for this calculation are simply the inverse of those stored in Table 6.2 and can be performed by performing a subtraction rather than addition for the two's complement bit. The computation can either be performed using parallel hardware or sequentially, by rotating the computation around an adder, as shown in Figure 6.7 where the final stage of computation is a subtraction rather than addition. It is clear to see that this computation can be performed using the basic CLB structure of the Xilinx FPGA family and the LE from Altera where the 4-bit LUT is used to store the DA data; the fast adder is used to perform the addition and the data is stored using the flip-flop. In effect, a CLB can perform one 'bit' of the computation meaning that now 8 LUTs are only needed to perform the computation admittedly at a slower rate than a parallel structure, due to the sequential nature of the computation.

Table 6.2 LUT contents for DA computation

Address				LUT contents
x_3^0	x_2^0	x_1^0	x_0^0	
0	0	0	0	0
0	0	0	1	a_0
0	0	1	0	a_1
0	0	1	1	$a_1 + a_0$
0	1	0	0	a_2
0	1	0	1	$a_2 + a_0$
0	1	1	0	$a_2 + a_1$
0	1	1	1	$a_2 + a_1 + a_0$
1	0	0	0	a_3
1	0	0	1	$a_3 + a_0$
1	0	1	0	$a_3 + a_1$
1	0	1	1	$a_3 + a_1 + a_0$
1	1	0	0	$a_3 + a_2$
1	1	0	1	$a_3 + a_2 + a_0$
1	1	1	0	$a_3 + a_2 + a_1$
1	1	1	1	$a_3 + a_2 + a_1 + a_0$

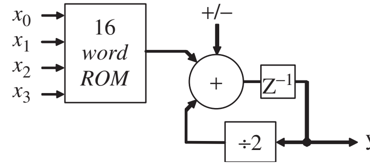


Figure 6.7 DA-based multiplier block diagram

$$\sum_{i=0}^{N-1} a_i(x_i^0) = a_0(x_0^0) + a_1(x_1^0) + a_2(x_2^0) + a_3(x_3^0) \tag{6.7}$$

$$\sum_{i=0}^{N-1} a_i(-x_i^0) = x_0^0(-a_0) + x_1^0(-a_1) + x_2^0(-a_2) + x_3^0(-a_3) \tag{6.8}$$

It is clear to see the considerable advantages that this technique offers for a range of DSP functions where one part of the computation is fixed. This includes some fixed FIR and IIR filters, a range of fixed transforms, namely the DCT and FFT and other selected computations. The technique has been covered in detail elsewhere (Peled and Liu 1974, Meyer-Baese 2001, White 1989) and a wide range of applications notes are available from each FPGA vendor on the topic.

6.6 Reduced Coefficient Multiplier

The DA approach has enjoyed considerable success and has really been the focal point of using the earlier FPGA technologies in DSP computations. However, the main limitation of the techniques is that the coefficients must be fixed in order to achieve the area reduction gain. If some applications require full coefficient range then a full programmable multiplier must be used, which in earlier devices was costly as it had to be built of existing LUT and adder resources, but in more recent FPGA families has been provided in the form of dedicated DSP hardware in the Xilinx Virtex™-5 and the Altera Stratix® families. However, in some applications such as the DCT and FFT, there is the need for limited range of multipliers. This is illustrated for the DCT computation.

The DCT is an important transformation, widely employed in image compression techniques. The two-dimensional (2D) DCT works by transforming an $N \times N$ block of pixels to a coefficient set which relates to the spatial frequency content that is present in the block. It is expressed as follows:

$$y(k,l) = \alpha(k)\alpha(l) \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} x(n,m)c(n,k)c(m,k) \tag{6.9}$$

where $c(n,k) = cos(2n+1)\pi k/2N$ and $c(m,k) = cos(2m+1)\pi k/2N$ and indices k and l range from 0 to $N-1$ inclusive. The values $\alpha(k)$ and $\alpha(l)$ are scaling variables. Typically, the separable property of the function is exploited, to allow it to be decomposed into two successive 1D transforms; this is achieved using techniques such as row–column decomposition which requires a matrix transposition function between the two 1D transforms.

The equation for an N-point 1D transform, which relates an input data sequence $x(i), i = 0>N-1$, to the transformed values $Y(k), k = 0>N-1$, is given in Equations (6.10) and (6.11).

$$Y(0) = \alpha(0) \sum_{n=0}^{N-1} x(i) \tag{6.10}$$

$$Y(k) = \alpha(k) \sum_{n=0}^{N-1} x(i) cos\left[\frac{k\alpha(2i+1)}{2N}\right] \quad \forall k = 1, \ldots, N-1 \tag{6.11}$$

where $\alpha(0) = 1/\sqrt{N}$, otherwise $\alpha(k) = 2/\sqrt{N}$. Expanding out the equation into a matrix form, we get the following matrix vector computation.

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \\ Y_5 \\ Y_6 \\ Y_7 \end{bmatrix} = \begin{bmatrix} C_4 & C_4 & C_4 & C_4 & C_4 & C_4 & C_4 & C_4 \\ C_1 & C_3 & C_5 & C_7 & -C_7 & -C_5 & -C_3 & -C_1 \\ C_2 & C_6 & -C_6 & -C_2 & -C_2 & -C_6 & C_6 & C_2 \\ C_3 & -C_7 & -C_1 & -C_5 & C_5 & C_1 & C_7 & -C_3 \\ C_4 & -C_4 & -C_4 & C_4 & C_4 & -C_4 & -C_4 & C_4 \\ C_5 & -C_1 & C_7 & C_3 & -C_3 & -C_7 & C_1 & -C_5 \\ C_6 & -C_2 & C_2 & -C_6 & -C_6 & C_2 & -C_2 & C_6 \\ C_7 & -C_3 & C_3 & -C_1 & C_1 & -C_1 & C_5 & -C_7 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{bmatrix} \tag{6.12}$$

In its current form the matrix vector computation would require 64 multiplications and 63 additions to compute the Y vector. However, a lot of research work has been undertaken to reduce the complexity of the DCT by precomputing the input data in order to reduce the number of multiplications. One such approach proposed by Chen *et al.* (1977) leads to the realization of the form given in Equation (6.13). These types of optimizations are possible and a range of such transformations exist for the DCT for both the 1D version (Hou 1987, Lee 1984, M.T. Sun *et al.* 1989) and the direct 2D implementation (Duhamel *et al.* 1990, Feig and Winograd 1992, Haque 1985, Vetterli 1985).

$$\begin{bmatrix} Y_0 \\ Y_2 \\ Y_4 \\ Y_6 \\ Y_1 \\ Y_3 \\ Y_5 \\ Y_7 \end{bmatrix} = \begin{bmatrix} C_4 & C_4 & C_4 & C_4 & 0 & 0 & 0 & 0 \\ C_2 & C_6 & -C_6 & -C_2 & 0 & 0 & 0 & 0 \\ C_4 & -C_4 & -C_4 & C_4 & 0 & 0 & 0 & 0 \\ C_6 & -C_2 & C_2 & -C_6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & C_1 & C_3 & C_5 & C_7 \\ 0 & 0 & 0 & 0 & C_3 & -C_7 & -C_1 & -C_5 \\ 0 & 0 & 0 & 0 & C_5 & -C_1 & C_7 & C_3 \\ 0 & 0 & 0 & 0 & C_7 & -C_5 & C_3 & -C_1 \end{bmatrix} \begin{bmatrix} X_0 + X_7 \\ X_1 + X_6 \\ X_2 + X_5 \\ X_3 + X_4 \\ X_0 - X_7 \\ X_1 - X_6 \\ X_2 - X_5 \\ X_3 - X_4 \end{bmatrix} \tag{6.13}$$

Equation (6.13) can either be implemented with a circuit where a fully programmable multiplier and adder combination could be used to implement either a row or a column, or possibly the whole circuit. However, this is unfortunate as the multiplier is only using 4 separate values at the multiplicand, as illustrated by the block diagram in Figure 6.8. This shows that ideally we need a multiplier which can cope with 3 separate values. Alternatively, a DA approach could be used where either 32 separate DA multipliers could be used to implement the matrix computation. Alternatively, 8 DA multipliers could be used, thereby achieving a reduction in hardware, but the dataflow would

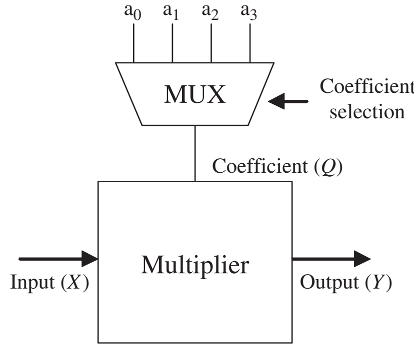


Figure 6.8 Reduced complexity multiplier

be complex in order to load the correct data at the correct time. It would be much more attractive to have a multiplier of the complexity of the DA multiplier which would multiply a limited range of multiplicands, thereby trading hardware complexity off with computational requirements which is exactly what is achieved in the RCM multipliers developed by Turner and Woods (2004).

6.6.1 RCM Design Procedure

The previous section on DA highlighted how the functionality could be mapped into a LUT-based FPGA technology. In effect, if you view the multiplier as a structure that generates the product terms and then uses an adder tree to sum the terms to produce a final output, then the impact of having fixed coefficients and organizing the computation as proposed in DA allows one to map a large level of functionality of the product term generator and adder tree within the LUTs. In essence, this is where the main area gain is achieved.

The concept is illustrated in Figure 6.9, although it is a little bit of illusion, as the actual AND and adder operations are not actually generated in hardware, but will have been precomputed. However, this gives us an insight in how we can map additional functionality onto LUT-based FPGAs which is the core part of the approach.

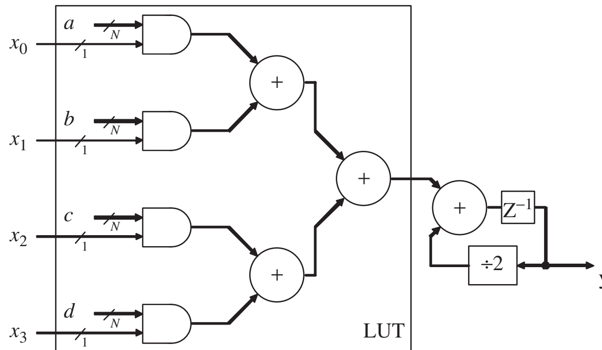


Figure 6.9 DA-based multiplier block diagram

In the DA implementation, the focus was to map as much as possible of the adder tree into the LUT. If we consider mapping a fixed coefficient multiplication into the same CLB capacity, then the main requirement is to map the EXOR function for the fast carry logic into the LUT as shown in Figure 6.10. This will not be as efficient as the DA implementation, but now the spare inputs can be used to implement additional functionality, as shown by the various structures of Figure 6.11. This is the starting point for us to create the structures for realizing a plethora of circuits.

Figure 6.11(a) implements the functionality of $A + B$ or $A + C$, Figure 6.11(b) implements the functionality of $A - B$ or $A - C$, and Figure 6.11(c) implements the functionality of $A + B$ or $A - C$. This leads to the concept of the generalized structure of Figure 6.12 and Turner (2002) gives a detailed treatment of how to generate these structures automatically, based on an input of desired coefficient values. However, here we use an illustrative approach to show the functionality of the DCT example is mapped using his technique, but it should be noted that this is not the technique used in Turner (2002).

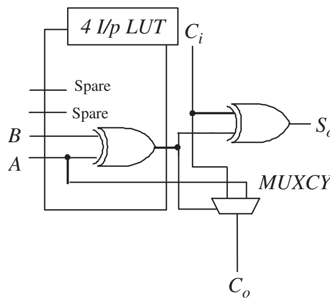


Figure 6.10 Mapping multiplier functionality into Virtex 4 CLB

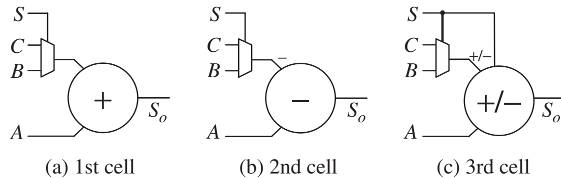


Figure 6.11 Possible implementations using the multiplexer-based design technique

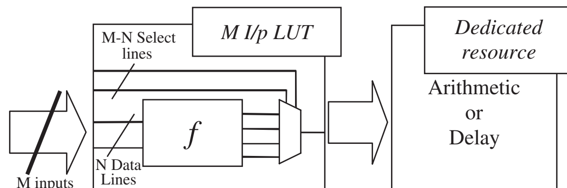


Figure 6.12 Generalized view of technique where a set of N input functions are mapped to an M input LUT and the spare line $M - N$ inputs are used for function selection

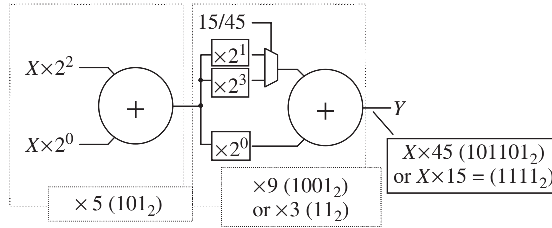


Figure 6.13 Multiplication by either 45 or 15

Some simple examples are used to demonstrate how the cells given in Figure 6.11, can be connected together to build multipliers. The circuit in Figure 6.13 multiplies, an input by two coefficients namely, 45 and 15. The notation $x \times 2^n$ represents a left-shifting by n. The circuit is constructed from two $2^n \pm 1$ cascaded multipliers taking advantage of $45 \times x$ and $15 \times x$ having the common factor of $5 \times x$. The first cell performs $5 \times x$ and then the second cell performs a further multiplication of 9 or 3, by adding either a shifted version by $2(2^1)$ or by $8(2^3)$, depending on the multiplexer control signal setting (labelled 15/45). The shifting operation does not require any hardware as it can be implemented as routing in the FPGA.

Figure 6.14 gives a circuit for multiplying by 45 or the prime number, 23. Here a common factor cannot be used, so a different factorization of 45 is applied, and a subtracter is used to generate multiplication by 15, i.e. $(16 - 1)$, needing only one operation as opposed to three. The second cell is set up to add a multiple of the output from the first cell, or a shifted version of the input X. The resulting multipliers implement the two required coefficients in the same area as a KCM for either coefficient, without the need for reconfiguration. Furthermore, the examples give some indication that there are a number of different ways of mapping the desired set of coefficients and arranging the cells in order to obtain an efficient multiplier structure.

In Turner and Woods (2004), the authors have derived a methodology for achieving the best solution, for the particular FPGA structure under consideration. The first step involves identifying the full range of cells of the type shown in Figure 6.11. Those shown only represent a small sample for the Xilinx Virtex™-II range. The full range of cells depends on the number of LUT inputs and the dedicated hardware resource. The next stage is then to encode the coefficients to allow the most efficient structure to be identified which was shown to be signed digit (SD) encoding. The coefficients are thus encoded and resulting shifted signed digits (SSDs) then grouped to develop the tree structure for the final RCM circuit. Prototype C++ software was developed to automate this process.

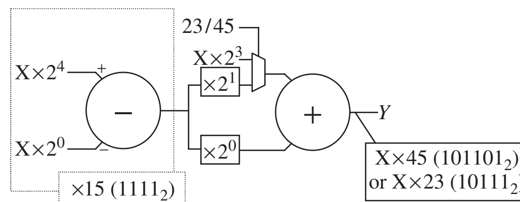


Figure 6.14 Multiplication by either 45 or 23

6.6.2 FPGA Multiplier Summary

The DA technique has been shown to work well in applications where the coefficients have fixed functionality. As can be seen from Chapter 2, this is not just limited to fixed coefficient operations such as fixed coefficient FIR and IIR filtering, but also could have application in fixed transforms such as the FFT and DCT. However, the latter RCM design technique provides a better solution as the main requirement is to develop multipliers that multiply a limited range of coefficients, not just a single value. The RCM technique has been demonstrated for DCT and a polyphase filter with a comparable quality in terms of performance to implementation based on DA techniques for other fixed DSP functions (Turner and Woods 2004, Turner *et al.* 2002).

It must be stated that changes in FPGA architectures, primarily the development of DSP48s in the Xilinx Virtex™-5 FPGA technologies, and the DSP function blocks in the latest Altera Stratix® III FPGA devices, have reduced the requirement to build fixed coefficient or even limited coefficient range structures as the provision of dedicated multiplier-based hardware blocks results in much superior performance. However, there may still be instances where FPGA resource is limited and these techniques, particularly if they are used along with pipelining, can result in implementations of the same speed performance of these dedicated blocks. Thus, it is useful to know that these techniques exist if required.

6.7 Final Statements

The chapter has aimed to cover some techniques that are specifically looking at mapping DSP systems onto specific FPGA platforms. Many will argue that in these days of improving technology and the resulting design productivity gap (IRTS 1999) we should move away from this aspect of the design approach altogether. Whilst the sentiment is well intentioned, there are many occasions where the detailed implementation has been important in realizing practical circuit implementations.

In image processing implementations as the design example in Figure 6.5 indicated, the derivation of a suitable hardware architecture is predicated on the understanding of what the underlying resources are, both in terms of speed and size. Thus a clear understanding of the practical FPGA limitations is important in developing a suitable architecture. Some of the other fixed coefficient techniques may be useful in applications where hardware is limited and users may wish to trade off between the DSP resource for other parts of the application. Whilst it has not been specifically covered in the chapter, a key aspect of efficient FPGA implementation is the development of efficient design styles. A good treatment of this was given by Michael Keating and Pierre Bricaud in their *Reuse Methodology Manual for System-On-A-Chip Designs* (Keating and Bricaud 1998). The main scope of this text was to highlight a number of good design styles that should be incorporated in the creation of efficient HDL code for implementation on SoC platforms. In addition to indications for good HDL coding, the text also offered some key advice on mixing clock edges, developing approaches for reset and enabling circuitry and clock gating. These are essential, but it was felt that the scope this book was to concentrate on the generation of the circuit architecture from a high level.

References

Chen WA, Harrison C and Fralick SC (1977) A fast computational algorithm for the discrete cosine transform. *IEEE Trans on Comms.* **COM-25**(9), 1004–1011.

Cocke J (1970) Global common subexpression elimination. *Proc. Symp. on Compiler Construction*, pp. 850–856.

Duhamel P, Guillemot C and Carlach J (1990) A DCT chip based on a new structured and computationally efficient DCT algorithm. *Proc. IEEE Int. Symp. on Circuits and Systems*, pp. 77–80.

Feher B (1993) Efficient synthesis of distributed vector multipliers. *Euromicro Symp. on Microprocessing and Microprogramming*, pp. 345–350.

Feig E and Winograd S (1992) Fast algorithms for the discrete cosine transform. *IEEE Trans. on Signal Processing* **40**(9), 2174–2193.

Goslin G (1995) Using xilinx FPGAs to design custom digital signal processing devices. *Proc. DSPX*, pp. 565–604.

Goslin G and Newgard B (1994) 16-tap, 8-bit FIR filter applications guide. Web publication downloadable from www.xiLinx.com.

Haque MA (1985) A two-dimensional fast cosine transform. *IEEE Trans. on Acoustics, Speech, and Signal Processing* **33**(6), 1532–1539.

Hou HS (1987) A fast recursive algorithm for computing the discrete cosine transform. *IEEE Trans. Acoustics, Speech, and Signal Processing* **35**(10), 1455–1461.

IRTS (1999) *International Technology* Roadmap for Semiconductors, 1999 edn. Semiconductor Industry Association. http://public.itrs.net

Keating M and Bricaud P (1998) *Reuse Methodology Manual for System-On-A-Chip Designs*. Kluwers, Norwell, MA, USA.

Lee BG (1984) A new algorithm to compute the discrete cosine transform. *IEEE Trans. on Acoustics, Speech and Signal Processing* **32**, 1243–1245.

Meyer-Baese U (2001) *Digital Signal Processing with Field Programmable Gate Arrays*. Springer, Germany.

Sun MT, Chen TC and Gottlieb AM (1989) VLSI implementation of a 16x16 discrete cosine transform. *IEEE Transaction on Circuits and Systems* **36**, 610–617.

Omondi AR (1994) *Computer Arithmetic Systems*. Prentice Hall, New York.

Peled A and Liu B (1974) A new hardware realisation of digital filters. *IEEE Trans. on Acoustics, Speech and Signal Processing*, pp. 456–462.

Turner RH (2002) *Functionally diverse programmable logic implementations of digital processing algorithms.* PhD dissertation, Queen's University Belfast.

Turner RH and Woods R (2004) Highly efficient, limited range multipliers for LUT-based FPGA architectures. *IEEE Trans. on VLSI Systems* **12**, 1113–1118.

Turner RH, Woods R and Courtney T (2002) Multiplier-less realization of a poly-phase filter using LUT-based FPGAs *Proc. Int. Conf. on Field Programmable Logic and Application*, pp. 192–201.

Vetterli M (1985) Fast 2-d discrete cosine transform. *Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, pp. 1538–1541.

White SA (1989) Applications of distributed arithmetic to digital siganl processing. *IEEE ASSP Magazine*, pp. 4–19.

Xilinx Inc. (2005) Using look-up tables as shift registers (srl-16) in spartan-3 generation FPGAs. Web publication downloadable from http://www.xilinx.com.

7

Rapid DSP System Design Tools and Processes for FPGA

7.1 Introduction

The evolution of computing architectures, spurred by the relentless growth in silicon integration technology, has seen the development of a number of new DSP implementation technologies. These platforms include single chip multiprocessor or heterogeneous system-on-chip solutions and indeed, FPGA. As highlighted as early as Chapter 1, the evolution in the computing architectures for this technology has for a number of years, outpaced the designers' ability to implement DSP systems using them. This observation has been popularly termed, the *design productivity gap* (IRTS 1999), and its main causes are a major limiting factor in the industry drive towards realising SoC design flows. It is forcing the electronic design automation (EDA) industry, to significantly reconsider the concepts of system design (Keutzer *et al*. 2000, Lee *et al*. 2003, Rowson and Sangiovanni-Vincentelli 1997).

Modern DSP implementation platforms are composed of a mixture of heterogeneous processing architectures, including microcontroller unit (MCU) Von Neumann-like processing architectures, increased computationally capable processors such as VLIW DSP microprocessors, or dedicated hardware for efficient task implementation. The evolution of the modern FPGA means that it is also a potential candidate as an implementation platform, either as a standalone SoPC, or as a complement to existing software-based platforms. FPGA-based embedded platforms propose entirely new and more complex implementation issues to the designer, due to the lack of a defined processing architecture.

This wide range of target processing architectures and corresponding implementation techniques makes DSP system implementation, at the current levels of design abstraction, an arduous task. Consequently, the use of coherent rapid implementation frameworks which translate a behavioural system description directly to an embedded manifestation, is critical. This chapter outlines current approaches to this problem.

Generally, the concept of model-based design of embedded systems is rapidly growing in popularity. This concept, a generalization of numerous specific design methodology types and tools, encourages the use of semantically well-defined modelling languages for expression of algorithm behaviour. The semantics of each of these types of models are then exploited to provide rapid implementation capabilities. As such, there are two crucial aspects to any design approach: the particular model of computation (MoC) employed for algorithm specification, and the methodology

FPGA-based Implementation of Signal Processing Systems R. Woods, J. McAllister, G. Lightbody and Y. Yi
© 2008 John Wiley & Sons, Ltd

used for refinement of these models leading to an implementation. This chapter outlines major developments in these fields in particular.

Following from this, FPGA-specific system synthesis problems are numerous. Three concerns are most prominent for the programmable logic community at present: synthesis of dedicated hardware intellectual property (IP) cores and core networks; software synthesis for multiprocessor architectures, and finally large-scale system level design for deriving and automatically realizing a given application on a heterogeneous FPGA architecture. These three aspects are addressed in turn, with a number of typical design tools in each field supplied.

The chapter is organized as follows. The reasons for the need for and prevalence of certain design approaches and tools are highly linked with the evolution of FPGA devices, and this motivation is outlined in Section 7.2. The underpinning design methodologies and modelling languages which enable toolsets such as those outlined above are outlined in Section 7.3. Section 7.4.3 describes how such modelling and rapid implementation techniques are exploited for single and multi-processor software synthesis; tools and techniques for IP core and core network synthesis are outlined in Section 7.5. Section 7.6 focuses on tools for automatically generating heterogeneous FPGA architectures and mapping algorithm specifications onto these architectures.

7.2 The Evolution of FPGA System Design

The unconventional evolutionary path charted by FPGA since the emergence of the first devices has strongly influenced the design methodology and tool requirements of modern devices. Three distinct ‘ages of FPGA’ can be identified, each of which has its own distinct device design and programming methods.

7.2.1 Age 1: Custom Glue Logic

When FPGAs first emerged, the relative dearth (by today’s standards) of logic resource on a single device means that little substantial functionality could be realized on a single chip, but their programmable nature meant FPGA made ideal host devices for customizable glue logic for multi-chip ASIC processing platforms. The low levels of complexity meant that single chip architectures could be realized with sufficiently high productivity using gate-level schematic-based capture of chip architectures and functionality. As such, gate-level schematic-based design prevailed during this period.

7.2.2 Age 2: Mid-density Logic

With the introduction of new generations of FPGA devices, such as the Xilinx XC4000 and Virtex devices and Altera Stratix devices, the levels of logic density realisable continued to increase with Moore’s Law. This, coupled with the potentially exceptionally high levels of parallelism meant that gradually FPGA moved from being simple glue logic enabling devices to hosts for complex DSP components such as filters and transforms. This evolution gave rise to an industry for the development of high-performance DSP IP cores. This trend continued with the introduction of high-performance small components such as multipliers, adders and memories on Virtex™ and Virtex™-2 series FPGAs from Xilinx and Cyclone devices from Altera. The increased design complexity for these devices means that the emergence of register transfer level (RTL) design tools such as Xilinx ISE and languages such as VHDL, complemented the emergence of RTL-level synthesis tools to translate EDIFs to FPGA programming files.

This phase resulted in a legacy of large numbers of IP cores and the associated need for design tools for core network construction have continued to play an important part in current FPGA design methodologies and tools. These are considered in Section 7.5.

7.2.3 Age 3: Heterogeneous System-on-chip

The emergence of Virtex™-2 Pro and Stratix FPGA devices meant that the traditional FPGA programmable logic was complemented by embedded microprocessors such as PowerPC in Virtex™-2 Pro, and high-speed dedicated serial communications transceivers such as Xilinx RocketI/O. Since then until the present, FPGA have been single chip heterogeneous processing solutions, with the performance capability and flexibility which meant they could be placed at the heart of DSP systems, rather than as add-on accelerators.

This substantial jump in device performance requires similarly substantial advances in FPGA design tool capabilities. The capability of these devices were now directly in league with SoC ASIC solutions, and whilst the performance of these devices was similar, it meant the FPGA design community had to adopt new technology in terms of multiprocessor software design, core network construction, heterogeneous system integration and high-speed off-chip and on-chip communications network design. Unfortunately, this technology was not readily forthcoming.

Whilst the move from between ages 1 and 2 of FPGA evolution was readily assisted by the industry-wide move from gate-level to RTL-level design abstraction, a similar standardization effort for the move from ages 2 to 3 has not been readily forthcoming. In the context of FPGA, such technology should address the issues of IP core synthesis and core network construction, multiprocessor system design, and system integration and optimization. Unfortunately, the initial design tool offerings, such as Xilinx Embedded Development Kit (EDK) and Altera SoPC Builder offered only primitive initial capabilities in this area. The remainder of this chapter outlines current state-of-the-art tools and practices in each of these areas, starting from the global system design methodology perspective (Section 7.3), and incorporating multiprocessor software synthesis (Section 7.4.3), IP core and core network synthesis (Section 7.5) and heterogeneous multiprocessor architecture design and synthesis is covered in Section 7.6.

7.3 Design Methodology Requirements for FPGA DSP

System design approaches have fallen largely into three main categories, namely: *hardware–software codesign* (HSC), *function–architecture codesign* (FAC - Fig.7.1(b) and *platform-based design* (PBD), Figure 7.1(a), (Keutzer *et al*. 2000).

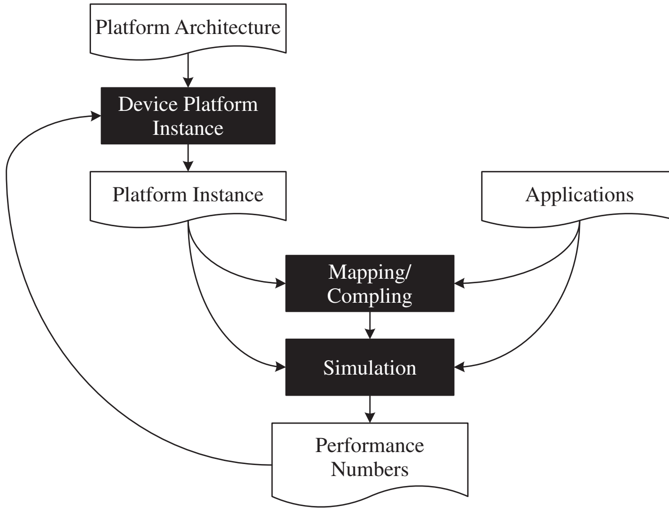
The PBD ethos follows the classic Y-chart approach to system design, where one of a domain of algorithms is mapped to a relatively fixed structure platform (although it may be tailored in a number of specific ways). The platform is considered a 'flexible' integrated circuit where customization for a particular application is achieved by 'programming' one or more of the components of the chip.

7.4 System Specification

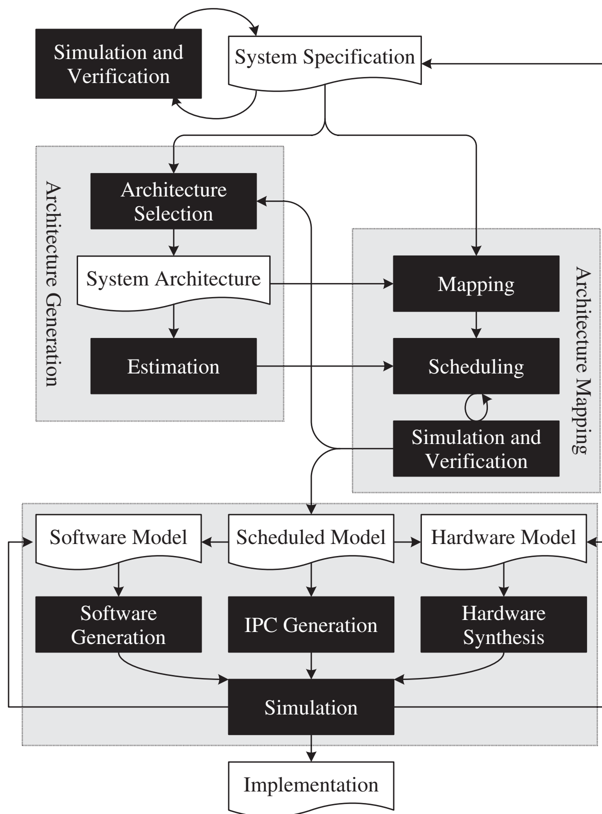
It should be noted from Section 7.1, that a key common aspect in all system design processes (in particular PBD and FAC), is the use of a model of the application in a well-defined model of computation (MoC) type language. In the description of design processes and tools which follow, it is apparent that these all follow this philosophy.

7.4.1 Petri Nets

A Petri net (Murata 1989) is a weighted, directed bipartite graph, consisting of *places* and *transitions* (Figure 7.2). Places (P_1–P_4 in Fig.7.2) contain *tokens*, with tokens moving from place to place via transitions. A transition (T_1 and T_2 in Figure 7.2) is enabled if each of its input places have a specified minimum number of tokens, at which point the specified numbers of tokens are removed from the input places, and a specified number inserted into the output places. The state of a Petri net is defined by a configuration of tokens in places, with the firing or transition rule of the net, determining the next state from the current state.



(a) PBD Y-Chart



(b) FAC

Figure 7.1 PBD and FAC design processes

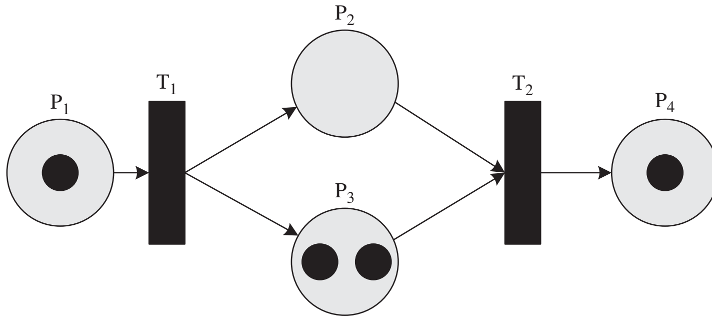


Figure 7.2 Petri net

7.4.2 Process Networks (PN) and Dataflow

The roots of the most popular current dataflow languages lie in the Kahn process network (KPN) model (Kahn 1974). The KPN model describes a set of parallel processes, or 'computing stations', communicating via unidirectional first-in first-out (FIFO) queues. A computing station consumes data tokens coming along its input lines, using localized memory, producing output on one or all of its output lines. In DSP systems, the tokens are usually digitized input data values. Continuous input to the system generates streams of input data, prompting the computing stations to produce streams of data on the system outputs. The general structure of a KPN is shown in Figure 7.3. The semantics of repetitive application of specific computing functions to every input sample in KPN makes this modelling approach a good match with the behaviour of DSP systems.

In the dataflow process network (DPN) model (Lee and Parks 1995), the KPN model is augmented with semantics for the computing station (known here as an actor) behaviour. A sequence of actor firings is defined to be a particular type of Kahn process called a dataflow process, where each firing maps input tokens to output tokens, and a succession maps input streams to output streams. A set of firing rules determine for each actor, how and when it fires. Specifically, actor firing *consumes* input tokens and *produces* output tokens. A set of sequential firing rules exist for each actor, and define the input data conditions under which the actor may fire. Given the solid computational foundation of DPN, which describes how actors fire and communicate deterministically, numerous application-specific refinements on this general theme have been proposed. Three specific variants are of particular importance in this section, synchronous dataflow (SDF), cyclo-static dataflow (CSDF)

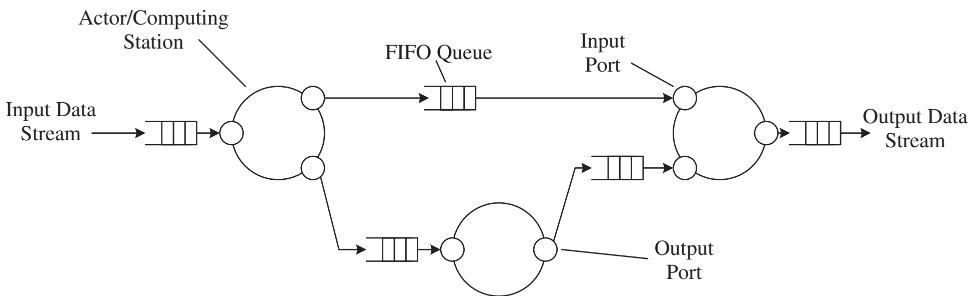


Figure 7.3 Simple KPN structure

and multidimensional synchronous dataflow (MSDF), variants whose capabilities are exploited in later chapters of this book.

As outlined previously, numerous different MoC languages are popularly exploited in modern system level design tools for embedded systems in general and FPGA in particular. Indeed, standardization efforts such as the *unified modelling language* or UML, (Marsh 2003) attempt to standardize the interaction of these models. UML has various domains of system description, each of which exposes different system characteristics and which are suitable for describing a different type of system. Whilst the implementations may be less efficient than hand-coded realizations in many specific systems cases, the rapid implementation still may produce adequately efficient results across a range of systems, saving on design time. A number of such approaches exploiting these techniques for software synthesis for single and multi-processor architectures are outlined next.

7.4.3 Embedded Multiprocessor Software Synthesis

A number of tools associated with UML, such as Real Time Studio from Artisan (Artisan Software Tools Ltd 2004) and Rhapsody from I-Logix can produce embedded code from such models. Other approaches use domain-specific graphical system descriptions. The MATLAB® Real Time Workshop (RTW) offers code generation capabilities for a number of target types directly from Simulink® graphical system descriptions. Given the availability of hardware synthesis flows from Simulink® for FPGA technology (Hwang *et al*. 2001), this places the MATLAB® toolsuite in a very promising position for heterogeneous system design, although an integrated heterogeneous environment does not yet exist.

Alternative approaches, such as the Rhapsody and Statemate tools from I-Logix (Gery *et al*. 2001, Hoffmann 2000) and Ptolemy Classic (Madahar *et al*. 2003), offer code generation capabilities from various domain-specific high-level system descriptions. Whilst each of these approaches offer specific rapid implementation capabilities, none is a complete system level heterogeneous FPGA design solution. A number of tools offer a different type of approach, where applications are described in a MoC (to provide a platform-independent description of the DSP algorithm), before interfacing to an API to enable rapid porting of autocoded implementations of the algorithm to the target platform. Typical of such an approach is GEDAE.

7.4.4 GEDAE

GEDAE is a graphical dataflow algorithm specification and embedded rapid multiprocessor implementation environment. The structure of the GEDAE development environment is shown in Figure 7.4(a). Every processor in the system is characterised by a board support package (BSP), which is the platform API. The flow graph editor connects a set of primitives (the basic leaf functions of the application) in a DFG format, and schedules them. The primitives are described as structured C functions with some imposed semantics to express the dataflow nature of the actors.

For high-level system optimization, a number of graph transformations can be applied, as will be described in Chapter 11. The DFG is partitioned across the processors in the platform, and a number of primitives inserted on-the-fly, such as inter-processor communication (send/recv) primitives, copy boxes for data movement within a schedule and sync primitives for synchronization of primitives with data streams. The resulting structure is then scheduled to generate the program to be executed on the target processor. For high-level system optimization, a number of graph transformations can be applied. The DFG is partitioned across the processors in the platform, and a number of inter-processor communication (*send*/*recv*) primitives are inserted on-the-fly. The resulting structure is then scheduled to generate the program to be executed on the target processor.

The result of the scheduler is a launch package consisting of executables for all of the microprocessors in the implementation platform and the schedule information for each processor. The

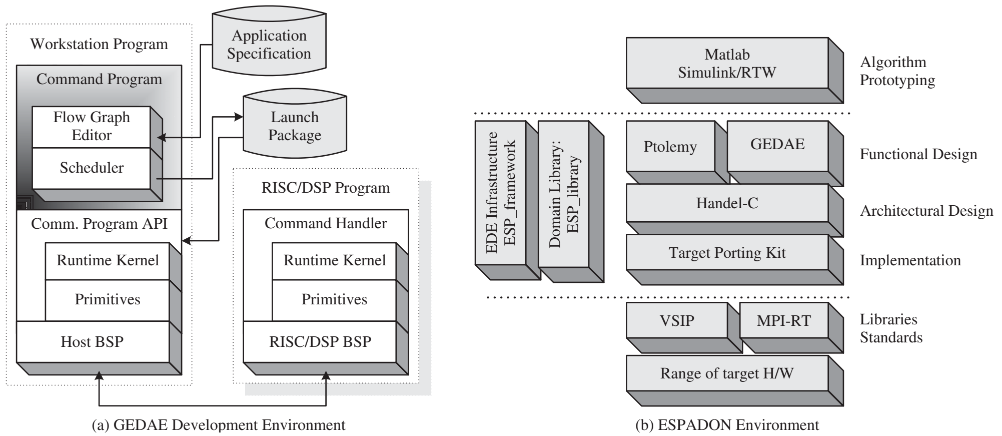


Figure 7.4 GEDAE in the ESPADON system-level design environment

schedules can utilize a set of optimized vector library functions, for improved function performance on embedded processors. The distributed application schedules for each processor are executed by the data driven runtime kernel. The command program on the host transfers data with the distributed application and has access to all elements of the distributed application for alteration and even dynamic reconfiguration. Part of the GEDAE BSP permits embedded inter-processor communication without the command program. When the application is deployed, the command program is removed, and the host system reads the launch package, initializes the various platform processors, and the application then runs distributed across the processors.

This currently plays a major part in current rapid prototyping approaches for military applications. Figure 7.4(b) shows the place of GEDAE in the ESPADON design flow, the result of a major European collaboration primarily involving BAE Systems and Thales.

7.5 IP Core Generation Tools for FPGA

7.5.1 Graphical IP Core Development Approaches

Block-based tools generate VHDL or Verilog HDL code from the block diagram to support hardware design. The code can then be fed to a hardware synthesis tool, to implement the DSP design in an FPGA or ASIC. The block-based approach still requires that the designer be intimately involved with the timing and control aspects of cores, in addition to being able to execute the back-end processor of the FPGA design flow. Furthermore, the only blocks available to the designer are the standard IP core library. The system designer must still be intimate with the underlying hardware in order to effectively implement the DSP algorithm into the hardware.

Most of these tools represent signal-processing algorithms on the popular Simulink® and MATLAB® platforms and enable FPGA implementation from these high-level descriptions, by generating VHDL or Verilog HDL code. Libraries of Simulink® blocks are provided to represent common signal processing functions. After users employ these special blocks to develop their algorithms, the FPGA tools can convert the design into an FPGA implementation. These tools include Xilinx's System Generator and Altera's DSP Builder.

System Generator (Xilinx Inc. 2000) uses the popular MATLAB®Simulink® tool from MathWorks Inc., and the cores developed by Xilinx Inc. to give a powerful high-level modelling environment which can be used for DSP system design. The algorithm is described using Simulink®, implemented initially using double precision arithmetic, then trimmed down to fixed point, and translated into hardware implementation. System Generator consists of a Simulink® library called the Xilinx Blockset, and software to translate a Simulink® model into a hardware realization of the model. In addition, System Generator automatically produces command files for FPGA synthesis, HDL simulation, and implementation tools, thereby allowing the user to work entirely in a graphical environment. The Xilinx Blockset can be freely combined with other Simulink® blocks, but only the Xilinx blocks and subsystems are converted into hardware. System Generator does this by mapping the Xilinx Blockset subsystems into IP library modules, and converting the Simulink® hierarchy into a hierarchical VHDL netlist.

Whilst this allows fast algorithmic level description, implementation issues such as core datapath latencies, levels of pipelining and numerical truncation, which can have a major impact on the resulting system performance, are not considered. Currently, it is left to the user to incorporate the impact of these issues when using the cores, and modify the model accordingly, which is not inconsiderable. Furthermore, the standard library of Xilinx IP Cores are the only blocks available to the designer. Other 'black-box' cores can be developed by a logic designer using standard HDL techniques, but these cannot currently be modelled in the same environment. When combined with the Xilinx System Generator™ for DSP tool, the new AccelDSP Synthesis 8.1 tool provides DSP algorithm and system designers who use MATLAB® and Simulink® design tools with the a capable design flow for high-performance DSP systems.

7.5.2 Synplify DSP

Synplify® DSP is an IP core architectural synthesis tool which generates RTL-level core architectures from combinations of low-level IP cores connected via the MATLAB Simulink® algorithm simulation environment. The designer develops their algorithm model in Simulink® and gradually refines it to a dedicated hardware FPGA architecture via a number of steps, as outlined in Figure 7.5. As this shows, the starting point is the ideal algorithm model, incorporating any manner of number representation system (such as float, double, etc.). This is then refined to a fixed-point equivalent in the Synplify DSP design environment. Each actor on the Simulink® graph is then mapped to an actor in a Simulink® blockset supplied with the tool, representing the behaviour of the core which represents that actor.

To account for discrepancies between the ideal algorithm behaviour and the physical behaviour of the equivalent core, a number of automated optimizations of the architecture are possible during the *DSP Synthesis* phase in Figure 7.5, including pipelining, retiming and automated vectorization of the architecture. The resulting finalized architecture is then converted to an RTL equivalent for implementation on the device via traditional RTL-level synthesis.

Further features, including fine-tuned design of state machine behaviour and architectures via synthesis from specification in a subset of MATLAB M-code, allow fine-grained optimization of the architecture and behaviour.

7.5.3 C-based Rapid IP Core Design

The C language is also commonly used to describe DSP algorithms because of the large amount of proven open source code that exists, particularly in standards-based applications. In addition,

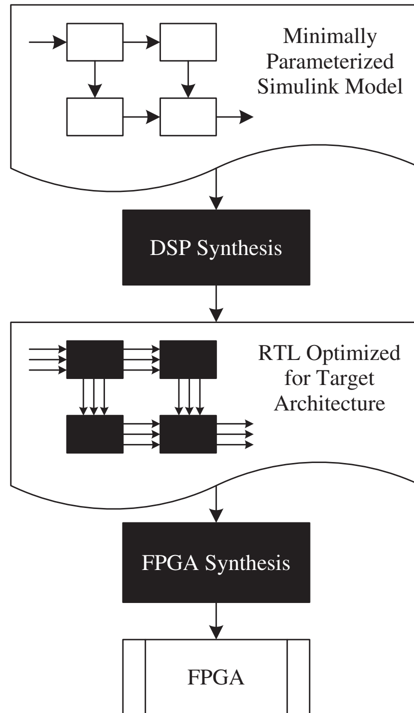


Figure 7.5 Synplify® DSP core synthesis

designing in C enables significant productivity gains when compared with a traditional Verilog or VHDL RTL design methodology due to the higher level of abstraction provided by C. Mentor Graphics Catapult C and Celoxica's Handel-C are two of the leading tools that enable designers to target FPGA product using C. C-based design tools for FPGAs have had their limitations in fully exploiting the FPGA architecture. FPGA constructs such as lookup tables, shift-register logic, and pipelining all need to be understood for implementing high-performance designs.

SPARK is a high-level synthesis framework for applying parallel compiler tranformations (Gupta *et al*. 2003). The effect of SPARK is that high-quality synthesis results are generated for designs with complex control flow. It takes an unrestricted input behavioural description in ANSI-C, parses it to give a hierarchical intermediate representation, and produces synthesizable RTL-level VHDL. During scheduling, a series of transformations are used including a data dependency extraction pass, parallelizing code motion techniques, basic operation of loop (or software) pipelining, and some supporting compiler passes (Gupta *et al*. 2003). After scheduling, the SPARK system then performs resource allocation, control synthesis and optimization to generate a FSM controller.

Handel-C is a high level programming language for implementing algorithms in hardware (FPGAs or ASICs), and the accompanying design tools allow architectural design space exploration, and hardware/software co-design (Oxford Hardware Compilation Group 1997), (Page 1996), (Celoxica Ltd 2002). It extends a small subset of C, removing processor-oriented features such as

pointers and floating-point arithmetic. It has been extended with a few constructs for configuring the hardware device and to support generation of efficient hardware. These hardware design extensions include flexible data widths, parallelism and communications between parallel elements.

Handel-C can be used to describe complex algorithms using all common expressions, and then translate them into hardware at the netlist level with a few hardware-optimizing features. Unfortunately, Handel-C allows only the design of digital, synchronous circuits and highly specialized hardware features are not provided. The low-level problems described earlier are hidden completely, because its focus is on fast prototyping and optimization at the algorithmic level, instead of investigation of all potentially possible design features. The compiler carries out all the gate-level decisions and optimization so that the programmer can focus on the design task.

7.5.4 MATLAB®-based Rapid IP Core Design

Although C/C++ has been the popular choice as a language for synthesis, users may choose MATLAB® in some approaches because of the following factors (Haldar 2001). First, MATLAB® provides a higher level of abstraction and less development time than C. Secondly, MATLAB® has a well-defined signal/image processing function blockset with a rich set of library functions related to matrix manipulation. In MATLAB®, the optimizations are easier and the compiler is less error-prone because of the absence of pointers and other complex data structures. Finally, extracting parallelism from MATLAB® is easier than C because automatic extraction of parallelism from C loops suffers from complex data dependency analysis, whereas DSP algorithms in MATLAB® are expressed as matrix operations which are very amenable to parallelization.

AccelFPGA (AccelFPGA 2002) provides a missing link between the DSP algorithm creation and FPGA hardware design. It produces optimized and synthesized RTL models from MATLAB® and Simulink® in terms of the target FPGA's internal execution resources, routing architecture and physical design. This designer productivity tool is based on MATCH (MATLAB® compiler for heterogeneous computing systems, (Haldar 2001)). It helps the designer to develop efficient code for configurable computing systems (Banerjee *et al*. 1999). In the MATCH project (Haldar 2001), a comparison in performance between compiler-generated hardware and manually designed hardware, indicates that the manually designed hardware was highly optimized for performance. Therefore, as AccelFPGA attempts to raise the abstraction level for design entry, higher levels of abstraction do not address the underlying complexities required for efficient implementations and so there is a trade-off between design time and quality.

7.5.5 Other Rapid IP Core Design

Synopsys Behavioral Compiler™ (Synopsys Inc. 2000) is a behavioural synthesis tool, which allows designers to evaluate alternative architectures, quickly generate an optimal gate-level implementation, and create designs that consist of a datapath, memory I/O and a control FSM. The features of Behavioral Compiler include chaining, multicycle operation, pipelined operations, loop pipelining, behavioural retiming and so on. Chaining schedules multiple intra-iteration-dependent operations into a single cycle, where the sum delay of the individual operators is less than the given clock period. Multicycle operations take more than a single cycle to complete, and are automatically scheduled over the required number of cycles. Behavioral Compiler can create multiple architectures from a single specification, and trade-off throughput and latency using high-level constraints because of these features.

MMAlpha, which is written in C and Mathematica, is a programming environment for manipulating and transforming the ALPHA program language (Derrien and Risset 2000). It provides a

path from high-level functional specification of an algorithm, to a synthesizable VHDL program. MMAlpha targets both ASICs and FPGAs, and is a functional data parallel language that allows expression of both recurrence equations and the hardware architecture of systolic arrays (Dupont de Dinechin *et al*. 1999). Advanced manipulation of ALPHA programs consists of pipelining, changes of basis, normalization and scheduling. Pipelining is a transformation widely used in systolic synthesis. It can be implemented by changing the basis in ALPHA. ALPHA also provides syntax for retiming synchronous architectures. In order to produce a hardware implementation for an ALPHA specification, ALPHARD, a subset of ALPHA, is used. ALPHARD enables a structural definition of a regular architecture such as systolic arrays to be given, and generates VHDL code at RTL level. However, only 1D and not 2D systolic arrays represented by ALPHARD programs can be translated to VHDL code.

JHDL (Bellows and Hutchings 1998) is a set of FPGA CAD tools based on Java HDL, originally for reconfigurable systems, developed at Brigham Young University. The main objective of JHDL is to develop a tool for describing circuits that can be dynamically changed over time. It allows the user to develop high-performance reconfigurable computing applications. JHDL is based on Java and is a structural design environment that results in smaller and faster circuits, compared with those developed using conventional behavioural synthesis tools. It supports some level of device independence, and uses TechMapper to target particular FPGA platforms. JHDL assumes a globally synchronous design, and does not support multi-clock synchronous and asynchronous loops. In addition, JHDL does not support behavioural synthesis. Compared with Handel-C, it is a lower level and provides much more control over the actual implementation in hardware, but is a much more difficult design environment.

7.6 System-level Design Tools for FPGA

7.6.1 Compaan

This research effort, is composed of three main tools, Compaan, LAURA and ESPAM. Compaan/LAURA (Stefanov *et al*. 2004) is a system-level design and optimization approach following the design ethics of PBD. Compaan is an automated toolset for generation of stream-based function (SBF) models (a variant of Process Networks with nodes of the structure shown in Figure 7.6(b) (Kienhuis and Deprettere 2001)) of DSP applications from nested loop representations written in conventional sequential languages such as MATLAB® or C++, as shown in Figure 7.6(a). The SBF algorithm representation can then be implemented as dedicated hardware networks on FPGA via an implementation paradigm, which produces wrappers for pre-design intellectual property cores, as described in Harriss *et al*. (2002). In recent years, this effort has expanded to produce the LAURA tool for implementation optimisation, and the ESPAM tool for code generation for multiprocessors communicating via a crossbar switch component. System optimization is performed largely by exploiting sequential code parallelization techniques such as loop unrolling or loop skewing, as outlined in Stefanov *et al*. (2002).

7.6.2 ESPAM

The ESPAM system synthesis tool (Nikolov *et al*. 2006) extends the design process proposed by Compaan to target heterogeneous multiprocessor platforms. Again, the design process starts with a sequential application specified in MATLAB® or C++ or an equivalent programing language, which is automatically converted to a parallel KPN specification by the KPNgen tool (Verdoolaege *et al*. 2007), which has added capabilities to process so-called weakly dynamic program types and estimate

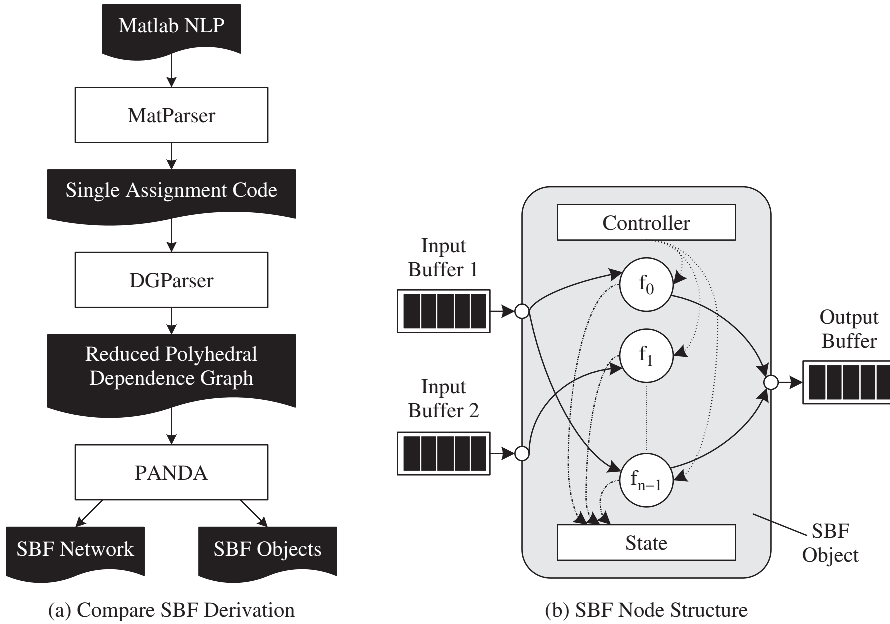


Figure 7.6 Compaan SBF network derivation

KPN FIFO buffer sizes. The resulting KPN is then mapped to a multiprocessor system-on-chip (MPSoC) platform specification. Source for each processor in the platform is generated, along with a crossbar, bus-based or point-to-point inter-processor communication architecture (Nikolov *et al.* 2006). System optimization is achieved via the application of various optimising transformations in the process of translating the input sequential specification to a KPN. An overview of the ESPAM design approach is given in Figure 7.7.

7.6.3 Daedalus

The Daedalus MPSoC design approach combines the synthesis capabilities of ESPAM with the system-level simulation capabilities of the Sesame system-level simulation and exploration environment, that has been developed at the University of Amsterdam (Pimentel *et al.* 2006, Thompson *et al.* 2007). Essentially, this combination augments the parallel algorithm derivation and rapid synthesis capabilities of ESPAM, with the capability to automatically explore the multiprocessor network topology and algorithm to architecture mapping design spaces. This allows automatic generation of the platform and mapping specifications to the ESPAM synthesis tool, where previously these inputs were manually generated. The overall Daedalus system design framework is shown in Figure 7.8. In Thompson *et al.* (2007), the developers have demonstrated the capabilities of this tool, to automatically explore the real-time performance design space for an MPEG application, by varying the number of PowerPC and Microblaze processor components in an MPSoC network, hosted on a Xilinx Virtex 2 Pro 2 FPGA.

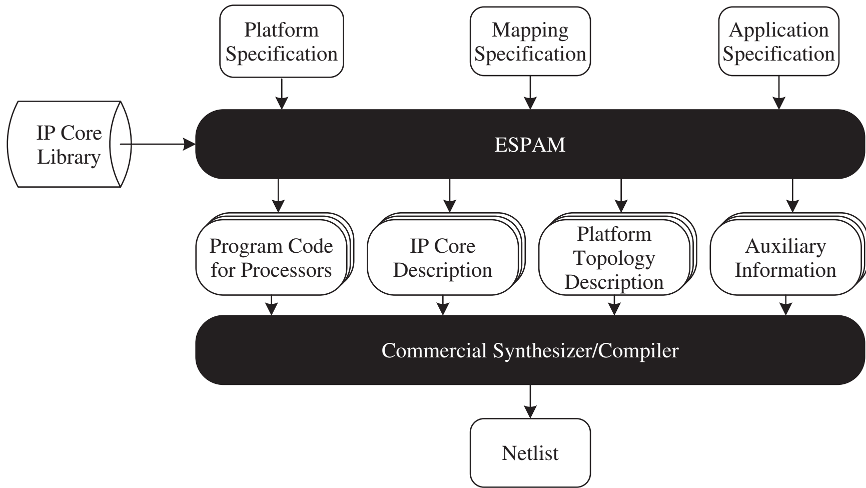


Figure 7.7 ESPAM MPSoC design process

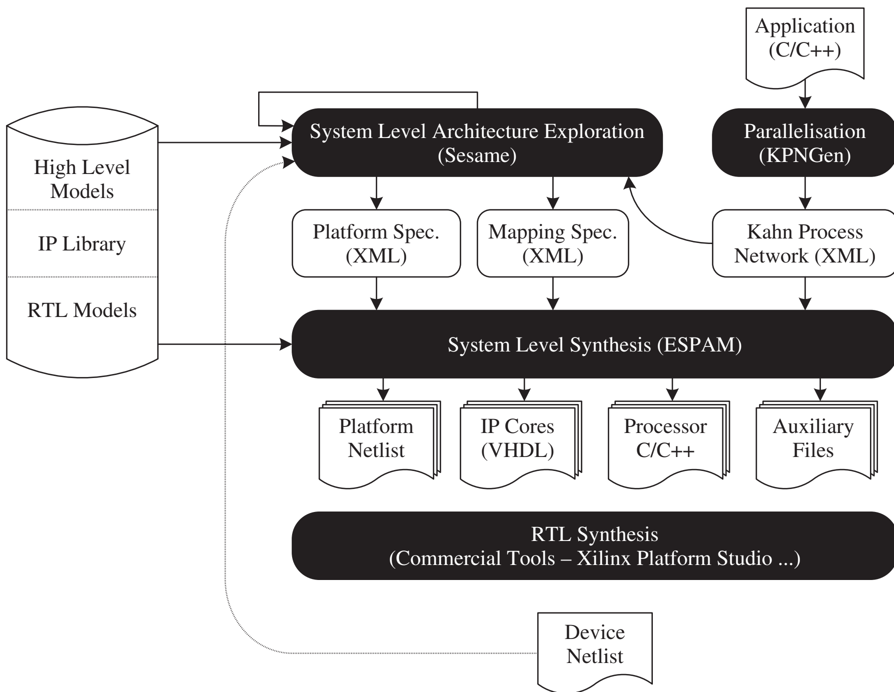


Figure 7.8 Daedalus MPSoC design process

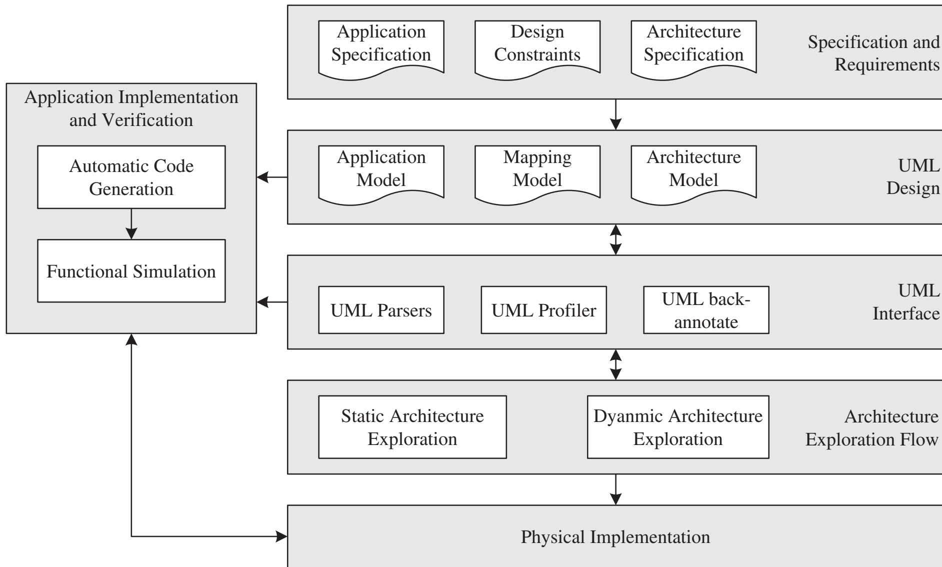


Figure 7.9 Koski MPSoC design environment

7.6.4 Koski

The Koski multiprocessor SoC (MP-SoC) design environment is a unified UML-based infrastructure for design space exploration and automatic synthesis, programming and prototyping of MPSoCs, primarily aimed at wireless sensor network applications (Kangas *et al*. 2006). The Koski environment structure is outlined in Figure 7.9.

Koski has a component-based design ethos, with the final architecture composed of numerous embedded processors and IP cores interface to a HIBI bus network (Salminen *et al*. 2006). As this shows, the inputs to Koski are manual specifications of application behaviour, the architecture and design constraints. Algorithm specification in Koski is via the KPN modelling language discussed in Section 7.4.2. From these, models of both the algorithm and architecture, as well as the mapping of KPN actors to the platform are derived. As Figure 7.9 shows, the architecture exploration consists of two phases: a *static* and a *dynamic* phase. The static phase determines the allocation of multiprocessor resources (in a variety of possible architectures from single to multiprocessor) and the mapping of the application onto the platform, and scheduling of the implementation. In the dynamic phase, finer-grained exploration is enabled, via time-accurate modelling of the implementation. Koski then supplies the software and RTL synthesis technology to enable the final implementation to be prototyped on an FPGA prototyping platform.

7.7 Conclusion

This chapter has outlined cutting-edge technology in the fields of application modelling, design methodology, IP core and core network synthesis, multiprocessor software synthesis and system-level design tools for heterogeneous FPGA systems. It has been shown that the rapid implementation capabilities of all these approaches stem from the exploitation of well-defined semantics in application-defining MoCs to produce embedded hardware and software implementations.

When considering rapid implementation tools in varying domains, it is notable that they have this single global similarity in common. Chapter 8 examines techniques for exploiting signal flow modelling for automatic, component-based synthesis of highly pipelined IP cores. It is revelatory to compare and contrast the treatment of the inherent dataflow modelling semantics, in this case with their treatment in multiprocessor software synthesis tools, such as GEDAE. Chapter 11 outlines how, although these two approaches have identical starting points, they are quite disparate and noncomplementary approaches. Furthermore, this chapter outlines techniques to exploit the common starting point for joint synthesis of systems which are more efficient than those which could be achieved using these current, disparate techniques.

References

AccelFPGA (2002) AccelFPGA: High-level synthesis for DSP design. Web publication downloadable from http://www.accelchip.com/.

Artisan Software Tools Ltd. (2004) Artisan real-time studio. Datasheet, available from http://www.artisansw .com/pdflibrary/Rts S.O datasheet.pdf.

Banerjee P, Shenoy N, Choudhary A, Hauck S, Bachmann C, Chang M, Haldar M, Joisha P, Jones A, Kanhare A, Nayak A, Periyacheri S and Walkden M (1999) Match: A MATLAB compiler for configurable computing system. Technical report, Center of Parallel and Distributed Computing, Northwestern University. CPDC-TR-9908-013.

Bellows P and Hutchings B (1998) JHDL – an HDL for reconfigurable systems. *IEEE Symposium on FPGA's for Custom Computing Machines*, pp. 175–184, Napa, USA.

Celoxica Ltd. (2002) Handle-C for hardware design. White Paper downloadable from http://www.celoxica.com.

Derrien S and Risset T (2000) Interfacing compiled FPGA programs: the MMAlpha approach. *Journal of Paralle and Distributed Processing Techniques and Applications*.

Dupont de Dinechin F, Quinton P, Rajopadhye S and Risset T (1999) First steps in ALPHA. *Publication Interne 1244, Irisa*.

Gery E, Harel D and Palachi E (2001) Rhapsody: A complete life-cycle model-based development system. *Proc. 3rd lat. conf. on Integrated Formal Methods*, Springer, pp. 1–10.

Gupta S, Dutt N, Gupta R and Nicolau A (2003) Spark: A high-level synthesis framework for applying parallelizing compiler transformations *Int. Conf. on VLSI Design*, New Delhi, pp. 461–466.

Haldar M (2001) *Optimised Hardware Synthesis for FPGAs* PhD Dissertation, Northwestern University.

Harriss T, Walke R, Kienhuis B and Deprettere EF (2002) Compilation from MATLAB to process networks realised in FPGA. *Design Automation for Embedded Systems* **7**(4), 385–403.

Hoffmann HP (2000) From concept to code: the automotive approach to using statemate magnum and rhapsody microC. White Paper available from http://whitepaper.silicon.com.

Hwang J, Milne B, Shirazi N and Stroomer JD (2001) System Level Tools for DSP in FPGAs. *Proc. 11th Int. Conf. on Field Programmable Logic and Applications*, pp. 534–5438.

IRTS (1999) *International Technology Roadmap for Semiconductors*, 1999 end. Semiconductor Industry Association. http://public.itrs.net

Kahn G (1974) The Semantics of a Simple Language for Parallel Programming *Proc. IFIP Congress*, pp. 471–475.

Kangas T, Kukkala P, Orsila H, E. S, Hannikainen B, Hamalainen T, Riihmaki J and Kuusilinna K (2006) Uml-based multiprocessor soc design framework. *ACM Transactions on Embedded Computing Systems (TECS)* **5**, 281–320.

Keutzer K, Malik S, Richard Newton S, Rabaey JM and Sangiovanni-Vincentelli A (2000) System level design: Orthogonolization of concerns and platform-based design. *IEEE Trans. CAD* **19**, 1523–1543.

Kienhuis B and Deprettere EF (2001) Modelling stream-based applications using the sbf model of computation *Proc. IEEE Workshop on Signal Processing Systems (SIPS)*, pp. 291–300.

Lee EA and Parks TM (1995) Dataflow Process Networks. *Proc. IEEE* **85**(3), 773–799.

Lee EA, Neuendorffer S and Wirthlin WJ (2003) Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems and Computers* **12**, 231–260.

Madahar BK, Alston ID, Aulagnier D, Schurer H and Saget B (2003) How rapid is rapid prototyping? Analysis of ESPADON programme results. *EURASIP Journal on Applied Signal Processing* **2003**(1), 580–593.

Marsh P (2003) Models of control. *IEE Elect. Syst. and Software* **1**(6), 16–19.

Murata T (1989) Petri nets: properties, analysis and applications. *Proc. IEEE* **77**(4), 541–580.

Nikolov H, Stefanov S and Deprettere E (2006) Multi-processor system design with ESPAM. *Proc. 4th Int. Conf. on Hardware/Software Codesign and System Synthesis*, pp. 211–216.

Oxford Hardware Compilation Group (1997) The Handel language. Technical report, Oxford University.

Page I (1996) Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing* **12**(1), 87–107.

Pimentel A, Erbas C and Polstra S (2006) A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transaction on Computers* **55**(2), 1–14.

Rowson JA and Sangiovanni-Vincentelli A (1997) Interface-based design *Proc. 34th Design Automation Conference*, pp. 178–183.

Salminen E, Kangas T, Hamalainen T, Riihimaki J, Lahtinen V and Kuusilinna K (2006) Hibi communication network for system-on-chip. *Proc. IEEE* **43**(2-3), 185–205.

Stefanov T, Kienhuis B and Deprettere E (2002) Algorithmic transformation techniques for efficient exploration of alternative application instances *Proc. 10th Int. Symp. on Hardware/Software Codesign*, pp. 7–12.

Stefanov T, Zissulescu C, Turjan A, Kienhuis B and Deprettere E (2004) System design using Kahn process networks: The Compaan/LAURA approach *Proc. Design Automation and Test in Europe (DATE) Conference*, p. 10340.

Synopsys Inc. (2000) Behavioural compiler™ quick reference. Web publication downloadable from http://www.synopsys.com/.

Thompson M, Stefanov T, Nikolov H, Pimentel A, Erbas C, Polstra E and Deprettere E (2007) A framework for rapid system-level exploration, synthesis and programming of multimedia MPSoCs *Proc. ACM/IEEE/IFIP Int. Conference on Hardware-Software Codesign and System Synthesis*, pp. 9–14.

Verdoolaege S, Nikolov H and Stefanov T (2007) Pn:a tool for improved derivation of process networks. *Eurasip Journal on Embedded Systems*. **1**.

Xilinx Inc. (2000) Xilinx system generator v2.1 for simulink reference guide. Web publication downloadable from http://www.xilinx.com.

8

Architecture Derivation for FPGA-based DSP Systems

8.1 Introduction

The technology review, Chapters 4 and 5, clearly demonstrated the need to develop a *circuit architecture* when implementing DSP algorithms on silicon hardware, whether the platform is ASIC or FPGA technology. The circuit architecture allows the performance needs of the application to be captured effectively. As was highlighted earlier, it is possible to implement high levels of parallelism available in the FIR filter expression (Equation 2.11 in Chapter 2), in order to achieve a performance increase, or to pipeline the SFG or dataflow graph (DFG) heavily. The first realization assumes that the hardware resource is available in terms of silicon area and the second approach assumes that the increased latency in terms of clock cycles, incurred as a result of the pipelining (admittedly at a smaller clock period), can be tolerated. It is clear that optimizations made at the hardware level can have direct cost implications for the resulting design. Both of these aspects can be captured in the circuit architecture.

As described in Chapter 5, this trade-off is much easier to explore in 'fixed architectural' platforms such as microprocessors, DSP processors or even reconfigurable processors, as sufficiently appropriate tools can, or have been developed to map the algorithmic requirements efficiently onto the available hardware. As already discussed, the main attraction of using FPGAs is that the high level of available hardware, can be developed to meet the specific needs of the algorithm. However, this negates the use of efficient compiler tools as in effect, the architectural 'goalposts' have moved as the architecture is created on demand! This fact was highlighted in Chapter 8 which covered some of the high-level tools that are being developed either commercially, or in universities and research labs. Thus, it is typical that a range of architecture solutions are explored with cost factors that are computed at a high level of abstraction.

In this chapter, we will explore the direct mapping of simple DSP systems or more precisely, DSP components such as FIR or IIR filters, adaptive filters, etc. as these will now form the part of more complex systems such as beamformers, echo cancellers, etc. The key focus is to investigate how changes applied to SFG representations can impact the FPGA realizations of such functions, allowing the reader to quickly work in the SFG domain, rather than at the circuit architecture domain. This trend becomes increasingly prevalent through the book, as we attempt to move to a higher level representation. The later chapters demonstrate how higher levels of abstraction can be employed to allow additional performance improvements by considering system level implications.

© 2008 John Wiley & Sons, Ltd

Section 8.2, looks at the DSP characteristics and gives some indication of how these map to FPGA implementation. We then attempt to describe how SFG changes are manifested in FPGA technology concentrating specifically on the Xilinx Virtex FPGA family. Given that a key aspect of FPGA architecture is the distributed memory, efficient pipelining is highlighted as a key optimization. This is explored in detail and formal methods to achieve the required levels of pipelining are presented. This section relies heavily on an excellent text by Keshab K. Parhi (Parhi 1999) which covers the implementation of complex DSP systems in VLSI hardware. The chapter then goes onto to cover a number of circuit level optimisations available to achieve the necessary speed but at a lower (or higher) area cost. This involves transforming the original SFG to achieve the necessary speed target; this could involve duplicating the hardware in order to improve the number of computations, a process called *unfolding*, or by sharing the available hardware if the speed available is too great. This is known as *folding*. Throughout the chapter, the techniques are applied to simple examples usually FIR and IIR filters.

8.2 DSP Algorithm Characteristics

By their very nature, DSP algorithms tend to be used in applications where there is a demand to process lots of information. As highlighted in Chapter 2, the sampling rates can range from kHz as in speech environments, right through to MHz, in the case of image processing applications. It is vital to clearly define a number of parameters with regard to system implementation of DSP systems:

- *sampling rate* can be defined as the rate at which we need to process the DSP signal samples for the system or algorithm under consideration. For example, in a speech application, the maximum bandwidth of speech is typically judged to be 4 kHz, resulting in a sampling rate of 8 kHz.
- *throughput rate* defines the rate at which data samples are processed. In some cases, the aim of DSP system implementation is to match the throughput and sampling rates, but in the lower sampling rates systems (speech and audio), this would result in under-utilization of the processing hardware. For example speech sampling rates are 8 kHz, but the speeds of many DSP processors are of the order of hundreds of MHz. In these cases, there is usually a need to perform a high number of computations per second which means that the throughput rate can be several of orders of magnitude higher say p, than the sampling rate. In cases where the throughput is high and the computational needs are moderate, then the possibility exists to reuse the hardware, say p times.
- *clock rate* defines the operating speed of the system implementation and is a figure that has been historically quoted by PC vendors to give some notion of performance. We argue it can be false in some applications as memory size, organization and usage can be more critical in determining performance. In DSP systems, a simple perusal of DSP and FPGA data sheets indicates that the clock rates of the latest Xilinx Virtex 5 FPGA family is 550 MHz whereas the TI's C64xx DSP family can run up to 1 GHz. At first thought, it would appear that the DSP processor is faster than the FPGA, but it is the amount of computation that can be performed in a single cycle that it is important. This is a major factor in determining the throughput rate which is a much more accurate estimate of performance, but is of course, application dependent.

Thus, it is clear that we need to design systems ultimately for throughput and therefore sampling rate, as a first measure of performance. This relies heavily on how efficiently we can develop the *circuit architecture*. As Chapters 4 and 5 clearly indicated, this comes from harnessing effectively the underlying hardware resources to meet the performance requirements. For ASIC applications, this is a fairly open hardware platform, but for FPGAs this is restricted in terms of processing

elements, e.g. dedicated DSP blocks, scalable adder structures, LUT resources, memory resource (distributed RAM, LUT RAM, registers) and interconnection, e.g. high-speed Rocket I/O, various forms of programmable interconnect etc. The aim is to match these resources to the computational needs which we will do here based initially on performance and then trading off area, if throughput is exceeded.

This does present major challenges. In DSP processors, the fixed nature of the architecture is such that efficient DSP compilers have evolved to allow high-level descriptions, for example C language descriptions, to be compiled, assembled and implemented onto the processors. Thus the aim of the DSP compiler is to investigate if the processing clock speed will allow one iteration of the algorithm to be computed at the required sampling rate. It does this by examining the fixed resources of the processor and scheduling the computation in such a way to achieve the required sampling rate. In effect, this involves reusing the available hardware, but we intend not to think about the process in these terms. From FPGA implementation, an immediate design consideration is to consider how many times can we reuse the hardware and does this allow us to achieve the sampling rate? This change of emphasis is creating the hardware resource to match the performance requirements is a key focus of the chapter.

8.2.1 Further Characterization

Some basic definitions of sampling, throughput and clock rates have now been explored for DSP systems. However, as we start to explore different techniques to change and improve circuit architectural descriptions, it is important to explore other timing aspects. For example, employing concurrency, in the form of parallelism and pipelining, can have additional impact on the performance and timing of the resulting FPGA implementations.

Latency is the time required to produce the output, $y(n)$ for the corresponding $x(n)$ input. At first glance, this would appear to equate to the throughput rate, but as the computation of $y(n) = a\,x(n)$ shown in Figure 8.1 clearly demonstrates, this is not the case, particularly if pipelining is applied. In Figure 8.1, the circuit could have three pipeline stages and thus will produce a first output after three clock cycles, hence known as the *latency*; thereafterwards, it will produce an output once every cycle which is the *throughput* rate. Consider the simple recursion $y(n) = a\,y(n-1)$ shown in Figure 8.2. The present output $y(n)$ is dependent on the previous output $y(n-1)$, and thus the

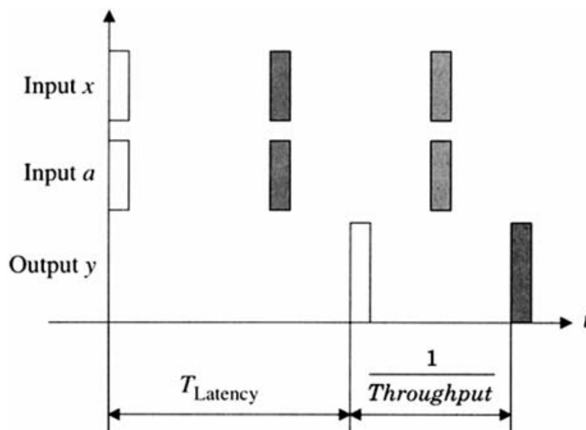


Figure 8.1 Latency and throughput rate relationship for system $y(n) = ax(n)$

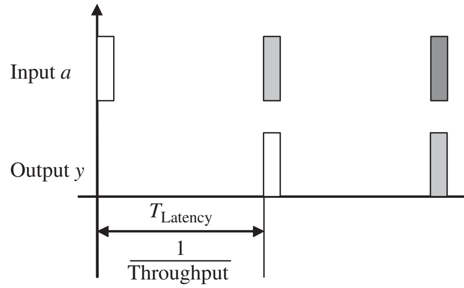


Figure 8.2 Latency and throughput rate relationship for system $y(n) = a, y(n-1)$

latency *determines* the throughput rate. This means that if it takes three clock cycles to produce the first output, then we have to wait three clock cycles for the circuit to produce each output and for that matter, enter every input. Thus it is clear that any technique such as pipelining that alters both the throughput and the latency, must be considered carefully, when deriving the circuit architectures for different algorithms.

There are a number of optimizations that can be carried out in FPGA implementation to perform the required computation, as listed below. Whilst it could be argued that *parallelism* is naturally available in the algorithmic description and not an optimization, the main definitions here focus around exploitation within *FPGA realization*; a serial processor implementation does not necessarily exploit this level of parallelism.

Parallelism can either naturally exist in the algorithm description or can be introduced by organizing the computation to allow a parallel implementation. In Figure 8.3, we can realize processes P_1, P_2 and P_3 as three separate processors, PE_1, PE_2 and PE_3 in all three cases. In Figure 8.3(a), the processes are driven from a single source, in Figure 8.3(b), they are from separate sources and in Figure 8.3(c), the processes are organized sequentially. In the latter case, the processing is inefficient as only one processor will be used at any one time, but it is shown here for completeness.

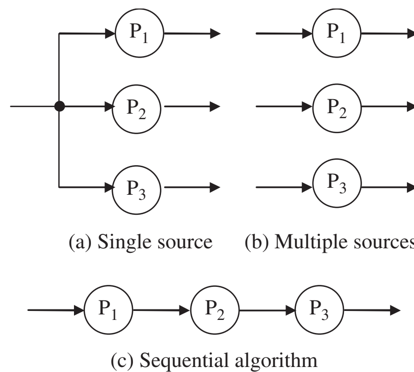


Figure 8.3 Algorithms realizations using three processors PE_1, PE_2 and PE_3

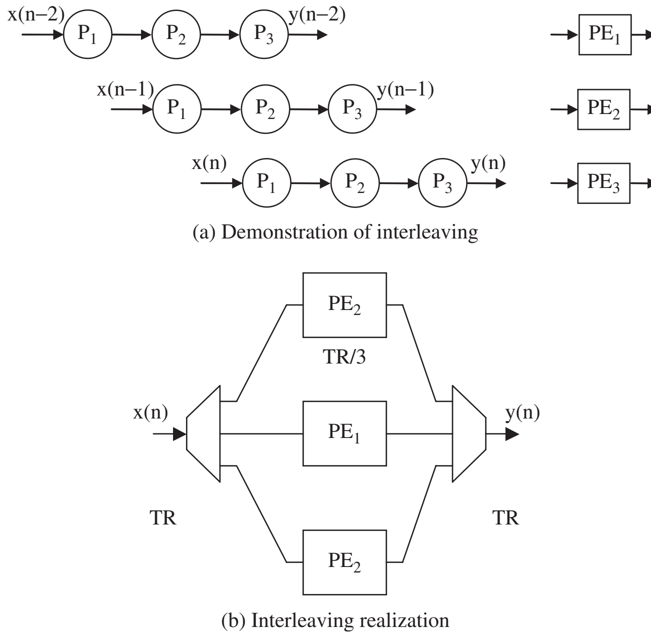


Figure 8.4 Interleaving example

Interleaving can be employed to speed up computation, by sharing a number of processors to compute iterations of the algorithm in parallel, as illustrated in Figure 8.4 for the sequential algorithm of Figure 8.3(c). In this case, the three processes P_1, P_2 and P_3 perform three iterations of the algorithm in parallel and each row of the outlined computation is mapped to an individual processor, PE_1, PE_2 and PE_3.

Pipelining is effectively another form of *concurrency* where processes are carried out on separate pieces of data, but at the same time, as illustrated in Figure 8.5. In this case, the three processes P_1, P_2 and P_3 are performed at the same time, but on different iterations of the algorithm. Thus the throughput is now given as $t\text{PE}_1$ or $t\text{PE}_2$ or $t\text{PE}_3$ rather than $t\text{PE}_1 + t\text{PE}_2 + t\text{PE}_3$ as for Figure 8.3(c). However, the application of pipelining is limited for some recursive functions such as the computation $y(n) = ay(n-1)$ given in Figure 8.6. As demonstrated in Figure 8.6(a), the original processor realization would have resulted in an implementation with a clock rate f_c and throughput rate f. Application of four levels of pipelining, as illustrated in Figure 8.6(b), results in an implementation that can be clocked four times faster, but since the next iteration depends on the present output, then it will have to wait four clock cycles. This gives a throughput rate of once every four cycles, indicating a nil gain in performance. Indeed, the flip-flop setup and hold times now form a much larger fraction of the critical path, and thus the performance would actually have been degraded in real terms.

It is clear then, whilst these optimizations exist, it is not a straightforward application of one technique. For example, it may be possible to employ parallel processing in the final FPGA

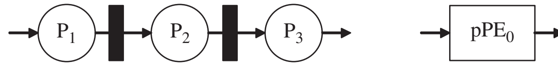
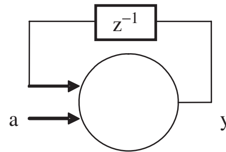
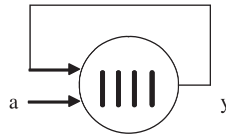


Figure 8.5 Example of pipelining



(a) Original recursive computation (clock rate, f and throughput rate, f)



(b) Pipelined version (clock rate, $4f$ and throughput rate, f)

Figure 8.6 Pipelining of recursive computations $y(n) = ay(n-1)$

realization and then employ pipelining within each of the processors. In Figure 8.6(b), pipelining did not give a speed increase, but now four iterations of the algorithm can be interleaved, thereby achieving a fourfold improvement. It is clear that there are a number of choices available to the designer to achieve the required throughput requirements with minimal area requirements such as sequential versus parallel, trade-off between parallelism/pipelining and efficient use of hardware sharing. The focus of this chapter is to demonstrate how the designer can start to explore these trade-offs in an algorithmic representation, by starting with a SFG or DFG description and then carrying out manipulations with the aim of achieving improved performance.

8.3 DSP Algorithm Representations

There are a number of ways of representing DSP algorithms ranging from mathematical descriptions, to block diagrams, right through to HDL descriptions of implementations. In this chapter, we concentrate on a SFG and DFG representation as we use this as a starting point for exploring some of the optimizations briefly outlined above. For this reason, it is important to provide more detail on SFG and DFG representations.

8.3.1 SFG Descriptions

The classical description of a DSP system is typically achieved using a signal flow graph (SFG). The SFG representation is a collection of nodes and directed edges, where a directed edge (j, k) denotes a linear transform from the signal at node j to the signal at node k. Edges are usually restricted to *multiplier*, *adder* or *delay* elements. The classical SFG of the expression $y(n) =$

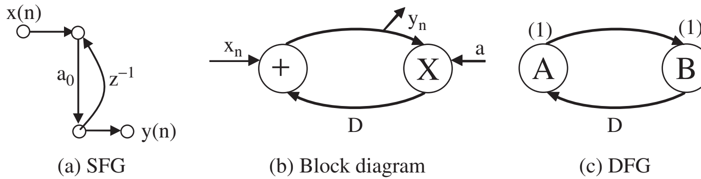


Figure 8.7 Various representations of simple DSP recursion $y(n) = ay(n-1) + x(n)$

$ay(n-1) + x(n)$ is given in Figure 8.7(a) whereas the block diagram is given in Figure 8.7(b). The DFG representation is shown in Figure 8.7(c) and it shown here, as it is a more useful representation for applying much of the retiming optimizations applied later in the chapter.

8.3.2 DFG Descriptions

In DFGs, nodes represent computations or functions and directed edges represent data paths with non-negative numbers associated with them. Dataflow captures the data-driven property of DSP algorithms where node can fire (perform its computation) when all the input data is available. This creates *precedence constraints* (Parhi 1999). There is an *intra-iteration constraint* if an edge has no delay, in other words the ordering of firing is dictated by DFG arrow direction. The *inter-iteration constraint* applies if the edge has one or more delays and will be translated to a digital delay or register when implemented.

A more practical implementation can be considered for a 3-tap FIR filter configuration. The SFG representation is given in Figure 8.8. One of the transformations that can be applied to SFG representation is that of *transposition*. This is carried out by reversing the directions in all edges, exchanging input and output nodes whilst keeping edge gains or edge delays unchanged as shown in Figure 8.8(b). The reorganized version is shown in Figure 8.8(c). The main difference is that the dataflow of the *x(n)* input has been reversed without causing any functional change to the resulting SFG. It will be seen later how the SFG of Figure 8.8(c) is a more appropriate structure to which to apply pipelining.

The data flow representation of the SFG of Figure 8.8(b) is shown in Figure 8.9. In Figure 8.9, the multipliers labelled as a_0, a_1 and a_2 represent pipelined multipliers with two levels of pipeline stages. The adders labelled as A_0 and A_1 represented pipelined adders with a pipeline stage of 1. The D labels represent single registers with size equal to the wordlength (not indicated on the DFG representation). In this way, the dataflow description gives a good indication of the hardware realization; it is clear that it is largely an issue of developing the appropriate DFG representation for the performance requirements needed. In the case of pipelined architecture, this is largely a case of applying suitable retiming methodologies to develop the correct level of pipelining, to achieve the performance required. The next section is totally dedicated to retiming because as will be shown there, recursive structures, i.e. those involving feedback loops, can present particular problems.

8.4 Basics of Mapping DSP Systems onto FPGAs

One of the main goals in attaining an FPGA realization is to determine the levels of pipelining needed. The timing of the data through the 3-tap FIR filter of Figure 8.8(a) for the nodes labelled

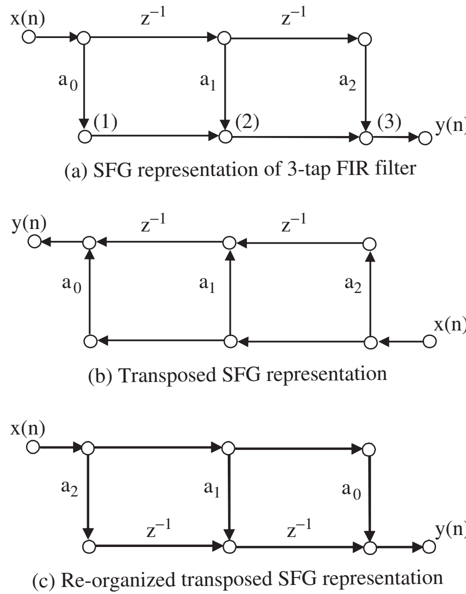


Figure 8.8 SFG representation of 3-tap FIR filter

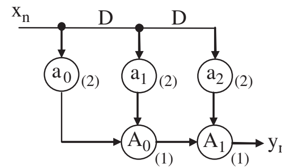


Figure 8.9 Simple DFG

(1), (2) and (3), is given in Table 8.1. We can add a delay to each multiplier output as shown in Figure 8.8(a), giving the change in data scheduling, as shown in Table 8.2. Note that the latency has now increased, as the result is not available for one cycle. However, adding another delay onto the outputs of the adders causes failure, as indicated by Table 8.3. This is because the process by which we are adding these delays has to be carried out in a systematic fashion by the application of a technique known as *retiming*. Obviously, retiming was applied correctly in the first instance as it did not change the circuit functionality but incorrectly in the second case. Retiming can be applied via the *cut theorem* as described in (Kung 1988).

8.4.1 Retiming

Retiming (Leiserson and Saxe 1983) is a transformation technique used to move delays in a circuit without affecting the input/output characteristics. Retiming has been applied in synchronous designs for clock period reduction (Leiserson and Saxe 1983), power consumption reduction (Monteiro *et al.*

Table 8.1 FIR filter timing

Clock	Input	Node 1	Node 2	Node 3	Output
0	$x(0)$	$a_0x(0)$	$a_0x(0)$	$a_0x(0)$	$y(0)$
1	$x(1)$	$a_0x(1)$	$a_0x(1) + a_1x(0)$	$a_0x(1) + a_1x(0)$	$y(1)$
2	$x(2)$	$a_0x(2)$	$a_0x(2) + a_1x(1)$	$a_0x(2) + a_1x(1) + a_2x(0)$	$y(2)$
3	$x(3)$	$a_0x(3)$	$a_0x(3) + a_1x(2)$	$a_0x(3) + a_1x(2) + a_2x(1)$	$y(3)$
4	$x(4)$	$a_0x(4)$	$a_0x(4) + a_1x(3)$	$a_0x(4) + a_1x(3) + a_2x(2)$	$y(4)$

Table 8.2 Revised FIR filter timing

Clock	Input	Node 1	Node 2	Node 3	Output
0	$x(0)$	$a_0x(0)$			
1	$x(1)$	$a_0x(1)$	$a_0x(0)$	$a_0x(0)$	$y(0)$
2	$x(2)$	$a_0x(2)$	$a_0x(1) + a_1x(0)$	$a_0x(1) + a_1x(0)$	$y(1)$
3	$x(3)$	$a_0x(3)$	$a_0x(2) + a_1x(1)$	$a_0x(2) + a_1x(1) + a_2x(0)$	$y(2)$
4	$x(4)$	$a_0x(4)$	$a_0x(3) + a_1x(2)$	$a_0x(3) + a_1x(2) + a_2x(1)$	$y(3)$

Table 8.3 Faulty application of retiming

Clock	Input	Node 1	Node 2	Node 3	Output
0	$x(0)$	$a_0x(0)$			
1	$x(1)$	$a_0x(1)$	$a_1x(0)$		
2	$x(2)$	$a_0x(2)$	$a_0x(1) + a_1x(0)$	$a_2x(0)$	$y(0)$
3	$x(3)$	$a_0x(3)$	$a_0x(2) + a_1x(1)$	$a_0x(1) + a_1x(0) + a_2x(0)$	
4	$x(4)$	$a_0x(4)$	$a_0x(3) + a_1x(2)$	$a_0x(2) + a_1x(1) + a_2x(1)$	

1993), and logical synthesis. The basic process of retiming is given in Figure 8.10 as taken from (Parhi 1999). For a circuit with two edges U and V and ω delays between them, as shown in Figure 8.10(a), a retimed circuit can be derived with ω' delays as shown in Figure 8.10(b), by computing the ω' value as given in equation (8.1) where $r(U)$ and $r(V)$ are the retimed values for nodes U and V, respectively.

$$\omega_r(e) = \omega(e) + r(U) - r(V) \tag{8.1}$$

Retiming has a number of properties which are summarized (Parhi 1999):

1. Weight of any retimed path is given by Equation (8.1).
2. Retiming does not change the number of delays in a cycle.

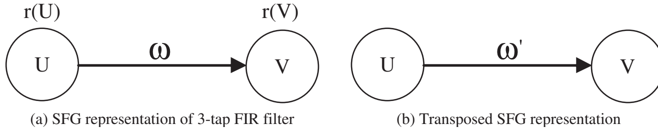


Figure 8.10 SFG representation of 3-tap FIR filter (Parhi 1999)

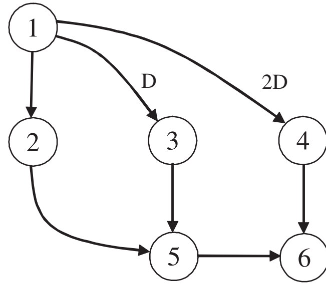
3. Retiming does not alter the iteration bound (see later) in a DFG as the number of delays in a cycle does not change.
4. Adding the constant value j to the retiming value of each node does not alter the number of delays in the edges of the retimed graph.

Figure 8.11 gives a number of examples of how retiming can be applied to the FIR filter DFG of Figure 8.11(a). For simplicity, we have replaced the labels of a_0, a_1, a_2, A_0 and A_1 of Figure 8.9 by 2, 3, 4, 5 and 6 respectively. We have also shown separate connections between the $x(n)$ data source and nodes 2, 3 and 4; the reasons for this will be shown shortly. By applying equation (8.1) to each of the edges, we get the following relationships for each edge:

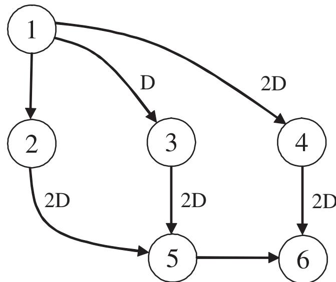
$$\begin{aligned}
\omega_r(1 \to 2) &= \omega(1 \to 2) + r(2) - r(1)\\
\omega_r(1 \to 3) &= \omega(1 \to 3) + r(3) - r(1)\\
\omega_r(1 \to 4) &= \omega(1 \to 4) + r(4) - r(1)\\
\omega_r(2 \to 5) &= \omega(2 \to 5) + r(5) - r(2)\\
\omega_r(3 \to 5) &= \omega(3 \to 5) + r(5) - r(3)\\
\omega_r(4 \to 6) &= \omega(4 \to 6) + r(6) - r(4)\\
\omega_r(5 \to 6) &= \omega(5 \to 6) + r(6) - r(5)
\end{aligned} \tag{8.2}$$

Using a retiming vector $r(1) = -2, r(2) = -2, r(3) = -2, r(4) = -2, r(5) = 0, r(6) = 0$ in Equation (8.2), we get the following values:

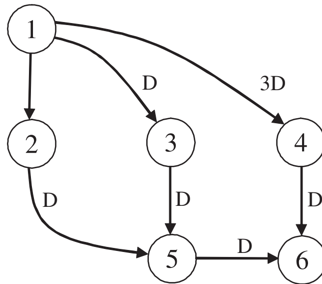
$$\begin{aligned}
\omega_r(1 \to 2) &= 0 + (-2) - (-2) = 0\\
\omega_r(1 \to 3) &= 1 + (-2) - (-2) = 1\\
\omega_r(1 \to 4) &= 2 + (-2) - (-2) = 2\\
\omega_r(2 \to 5) &= 0 + (0) - (-2) = 2\\
\omega_r(3 \to 5) &= 0 + (0) - (-2) = 2\\
\omega_r(4 \to 6) &= 0 + (0) - (-2) = 2\\
\omega_r(5 \to 6) &= 0 + (0) - (0) = 0
\end{aligned}$$



(a) DFG of original 3-tap FIR filter



(b) FIR filter DFG retimed with r(1) = −2, r(2) = −2, r(3) = −2
r(4) = −2, r(5) = 0, r(6) = 0



(c) FIR filter DFG retimed with r(1) = −2, r(2) = −2, r(3) = −2
r(4) = −1, r(5) = −1, r(6) = 0

Figure 8.11 Retimed FIR filter

This gives the revised diagram shown in Figure 8.11(b) which gives a circuit where each multiplier has two pipeline delays at the output edge. A retiming vector could have been applied which provides one delay at the multiplier output, but the reason for this retiming will be seen later. Application of an alternative retiming vector namely, $r(1) = -2, r(2) = -2, r(3) = -2$, $r(4) = -1, r(5) = -1, r(6) = 0$ gives the circuit of Figure 8.11(c) which gives a fully pipelined implementation. It can be seen from this figure that the application of pipelining to the adder stage

required an additional delay D, to be applied to the connection between 1 and 4. It is clear from these two examples that a number of retiming operations can be applied to the FIR filter. A retiming solution is feasible if $\omega_r \geq 0$, holds for all edges.

$$\omega_r(1 \rightarrow 2) = 0 + (-2) - (-2) = 0$$

$$\omega_r(1 \rightarrow 3) = 1 + (-2) - (-2) = 1$$

$$\omega_r(1 \rightarrow 4) = 2 + (-1) - (-2) = 3$$

$$\omega_r(2 \rightarrow 5) = 0 + (-1) - (-2) = 1$$

$$\omega_r(3 \rightarrow 5) = 0 + (-1) - (-2) = 1$$

$$\omega_r(4 \rightarrow 6) = 0 + (0) - (-1) = 1$$

$$\omega_r(5 \rightarrow 6) = 0 + (0) - (-1) = 1$$

It is clear from the two examples outlined that retiming can be used to introduce *inter-iteration constraints* (Parhi 1999) to the DFG which is manifested as a pipeline delay in the final FPGA implementation. However, the major issue would appear to be the determination of the retiming vector which must be such that it moves the delays to the edges needed in the DFG whilst at the same time, preserving the viable solution, i.e. $\omega_r \geq 0$ holds for all edges. One way of determining the retiming vector, is to apply a graphical methodology to the DFG which symbolizes applying retiming. This is known as the *cut-set* or *cut* theorem and was presented by Kung (1988).

8.4.2 Cut-set Theorem

A cut-set in an SFG (or DFG) is a minimal set of edges, which partitions the SFG into two parts. The procedure is based upon two simple rules.

Rule 1: *Delay scaling.* All delays D presented on the edges of an original SFG may be scaled, i.e., $D' \longrightarrow \alpha D$, by a single positive integer α, which is also known as the pipelining period of the SFG. Correspondingly, the input and output rates also have to be scaled by a factor of α (with respect to the new time unit D′). Time scaling does not alter the overall timing of the SFG.

Rule 2: *Delay transfer*. (Leiserson and Saxe 1983) Given any cut-set of the SFG, which partitions the graph into two components, we can group the edges of the cut-set into inbound and outbound, as shown in Figure 8.12, depending upon the direction assigned to the edges. The delay transfer rule states that a number of delay registers, say k, may be transferred from outbound to inbound edges, or vice versa, without affecting the global system timing.

Let's consider the application of Rule 2 to the FIR filter DFG of Figure 8.11(a). The first *cut* is applied in Figure 8.13(a) where the DFG graph is cut into two distinct regions or sub-graphs, *sub-graph # 1* comprising nodes 1, 2, 3 and 4, and *sub-graph # 2* comprising 5 and 6. Since all edges between the regions are outbound from *sub-graph # 1* to *sub-graph # 2*, a delay can be added to each. This gives Figure 8.11(b). The second *cut* splits the DFG into *sub-graph # 3*, comprising nodes 1, 2, 3 and 5 and *sub-graph # 4* comprising nodes 4 and 6. The addition of a single delay to this edge lead to the final pipelined design, as shown in Figure 8.11(c).

These rules provide a method of systematically adding, removing and distributing delays in a SFG and therefore adding, removing and distributing registers throughout a circuit, without changing the function. The cut-set retiming procedure is then employed, to cause sufficient delays to appear

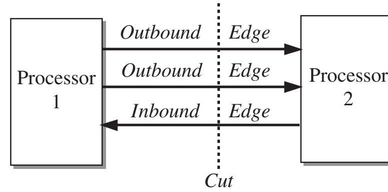


Figure 8.12 Cut-set theorem application

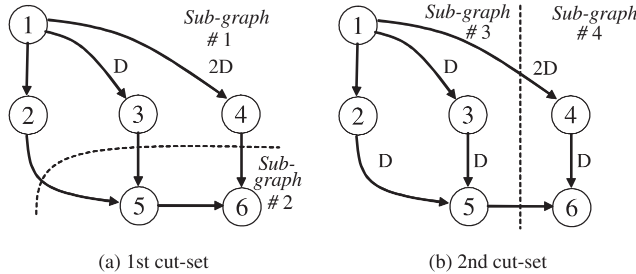


Figure 8.13 Cut-set timing applied to FIR filter

on the appropriate SFG edges, so that a number of delays can be removed from the graph edges and incorporated into the processing blocks, in order to model pipelining within the processors; if the delays are left on the edges, then this represents pipelining between the processors.

Of course, the selection of the original algorithmic representation can have a big impact on the resulting performance. Take, for example, the alternative version of the SFG shown initially in Figure 8.8(c), and represented as a DFG in Figure 8.14(a); applying an initial cut-set allows pipelining of the multipliers as before, but now applying the cut-set between nodes 3 and 5, and nodes 4 and 6, allows the delay to be transferred resulting in a circuit architecture with a lower number of delay elements as shown in Figure 8.14(c).

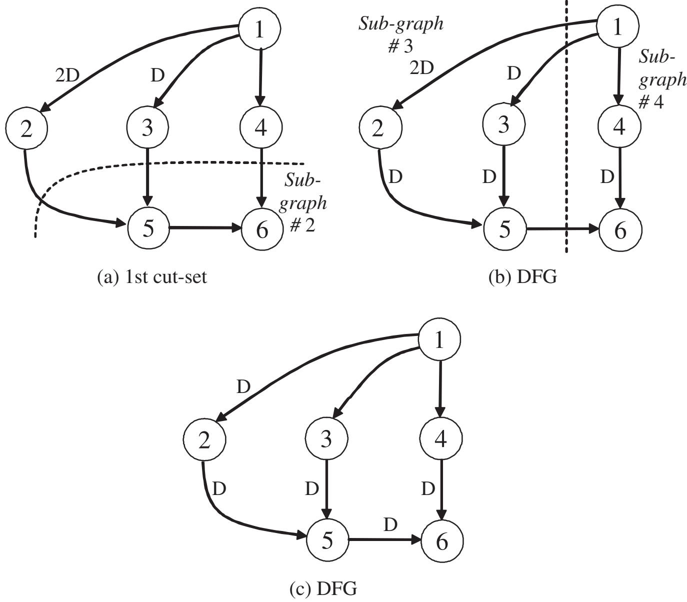
8.4.3 Application of Delay Scaling

In order to investigate the delay scaling aspect, let's consider a recursive structure such as the second order IIR filter section given in Equation (8.3). The block diagram and the corresponding DFG is given in Figures 8.15(a) and 8.15(b) respectively. The target is to apply pipelining at the processor level, thereby requiring a delay D on each edge. The problem is that there is not sufficient delays in the $2 \rightarrow 3 \rightarrow 2$ loop, to apply retiming. For example, if the cut shown on the figure was applied, this would end up moving the delay on edge $3 \rightarrow 2$ to edge $2 \rightarrow 3$. The issue is resolved by applying time scaling, by working out the worse case *pipelining period*, as defined by Equations (8.4) and (8.5).

$$y(n) = a_0x(n) + a_1x(n-1) + a_2x(n-2) + b_1y(n-1) + b_2y(n-2) \tag{8.3}$$

$$\alpha_C = \frac{B_c}{D_c} \tag{8.4}$$

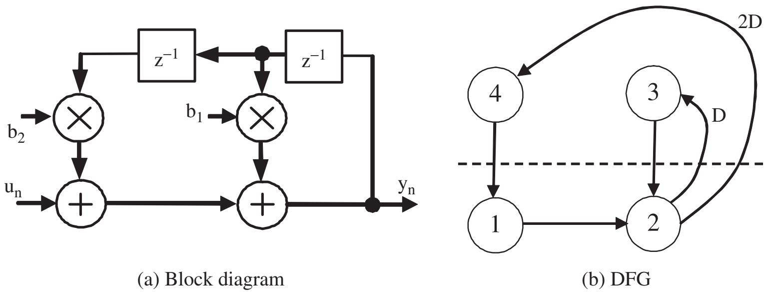
$$\alpha = \max\{all\alpha_c\} \tag{8.5}$$



(a) 1st cut-set (b) DFG

(c) DFG

Figure 8.14 Cut-set timing applied to FIR filter



(a) Block diagram (b) DFG

Figure 8.15 2nd-order IIR filter

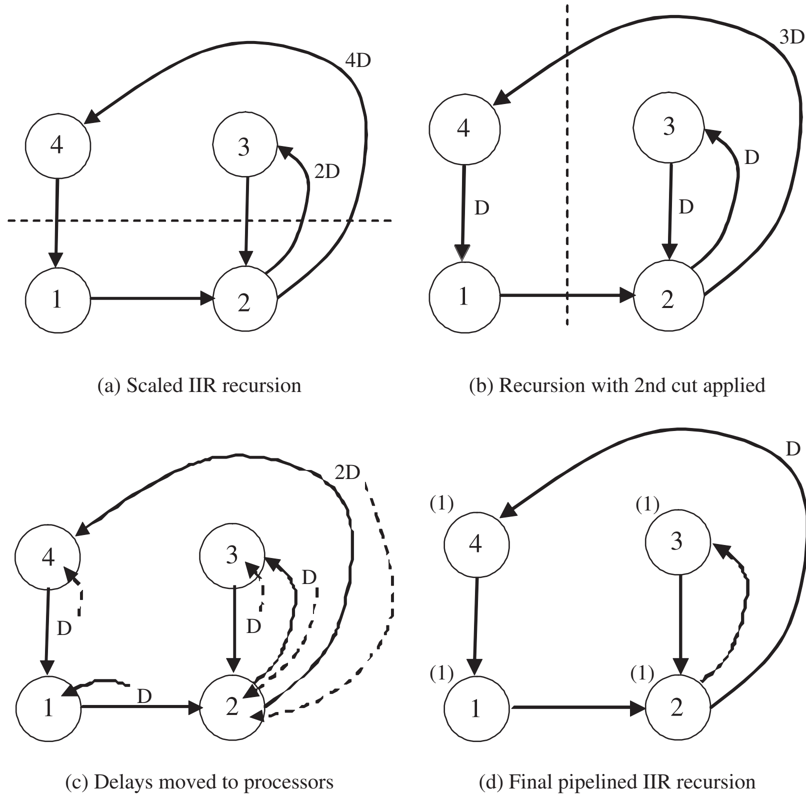
In Equation (8.4), the value B_c refers to the delays required for processor pipelining and the value D_c refers to the delays available in the original DFG. The optimal pipelining period is computed using Equation (8.5) and is then used as the scaling factor. There are two loops as shown, giving a worst-case loop bound of 2. The loops are given in terms of unit time (u.t.) steps.

$$1 \rightarrow 2 \rightarrow 4 \rightarrow 1 \text{ (3 u.t.)}$$

$$2 \rightarrow 3 \rightarrow 2 \text{ (2 u.t.)}$$

$$\text{Loop bound} \sharp 1 \ (3/2 = 1.5 \text{ u.t.})$$

$$\text{Loop bound} \sharp 2 \ (2/1 = 2 \text{ u.t.})$$



(a) Scaled IIR recursion
(b) Recursion with 2nd cut applied
(c) Delays moved to processors
(d) Final pipelined IIR recursion

Figure 8.16 Pipelining of a 2nd-order IIR filter

Table 8.4 Faulty application of retiming

	Area		Throughput	
Circuit	DSP48	Flip-flops	Clock (MHz)	Data rate (MHz)
Figure 8.15(b)	2	20	176	176
Figure 8.16(d)	2	82	377	188

The process of applying the scaling and retiming is given in Figure 8.16. Applying scaling of 2 gives the retimed DFG of Figure 8.16(a). Applying the cut shown in the figure gives the modified DFG of Figure 8.16(b) which then has another cut applied, giving the DFG of Figure 8.16(c). Mapping of the delays into the processor and adding the numbers to show the pipelining level gives the final pipelined IIR recursion (Figure 8.16(d)).

The final implementation has been synthesized using the Xilinx Virtex 5 FPGA and the synthesis results can be viewed for the circuits of Figure 8.15(b) and Figure 8.16(d) in Table 8.4.

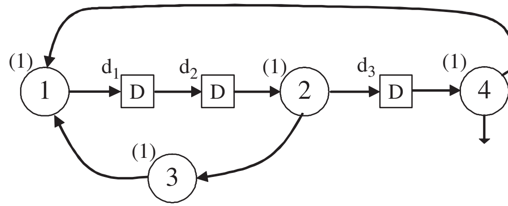


Figure 8.17 Simple DFG example (Parhi 1999)

8.4.4 Calculation of Pipelining Period

The previous sections have outlined a process for first determining the pipelining period then allowing scaling of this pipelining period to allow pipelining at the processor level which is the finest level of pipelining possible within FPGA technology (Although, as will be seen in Chapter 14, adding higher levels of pipelining can be beneficial for low-power FPGA implementations). However, the computation of the pipelining period was only carried out on a simple example of an IIR filter 2nd-order section and therefore much more efficient means of computing the pipelining period are needed. A number of different techniques have been presented in (Parhi 1999). The one considered here is known as the *longest path matrix* algorithm (Parhi 1999) and is best illustrated with an example.

Longest Path Matrix Algorithm

A series of matrices is constructed and the iteration bound is found by examining the diagonal elements. If d is the number of delays in DFG, then create $L^{(m)}$ matrices where $m = 1, 2, >, d$ such that element $l_{1,j}^{1}$ is the longest path from delay element d_i which passes through exactly $m - 1$ delays (not including d_i and d_j); if no path exists, then $l_{1,j}^{1}$ is -1. The longest path can be computed using Bellman–Ford or Floyd–Warshall algorithm (Parhi 1999).

Example 1. Consider the example given in Figure 8.17. Since the aim is to produce a pipelined version of the circuit, we have started with pipelined version indicated by the (1) expression included in each processor. This can be varied by changing the expression to (0) if the necessary pipelining is not required, or to a higher value, e.g. (2) or (3), if additional pipelined delays are needed in the routing to aid placement and routing or for low-power implementation.

The first stage is to compute the $L^{(m)}$ matrices, beginning with the $L^{(m)}$ matrix. This is computed by generating each term, namely $l_{1,j}^{1}$, which is given as the path from delay d_i through to d_j. For example, d_1 to d_1 passes through either 1 ($d_1 \rightarrow d_2 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow d_1$) or 2 delays ($d_1 \rightarrow d_2 \rightarrow 2 \rightarrow d_4 \rightarrow 1 \rightarrow d_1$), therefore $(1, 1) = -1$. For $l_{3,1}^{1}$, the path d_3 to d_1 passes goes through nodes (4) and (1), giving a delay of 2; therefore, $l_{3,1}^{1} = 2$. For $l_{2,1}^{1}$, the path d_2 to d_1 passes goes through nodes (2), (3) and (1), therefore $l_{2,1}^{1} = 3$. This gives the matrix as shown below:

$$\begin{pmatrix} -1 & 0 & -1 \\ 7 & -1 & 3 \\ 3 & -1 & -1 \end{pmatrix}$$

The higher-order matrices do not need to derived from the DFG. They can be recursively computed as:

$$l_{i,j}^{m+1} = \overset{\max}{k\varepsilon K} \left(-1, l_{i,j}^{1} + l_{k,j}^{m}\right)$$

where K is the set of integers k in the interval $[1, d]$ such that, neither $l_{i,k}^{1} = -1$ nor $l_{i,k}^{m} = -1$ holds. Thus for $l_{1,1}^{2}$, we can consider $K = 1, 2, 3$ but 1,3 include -1, so only $K = 2$ is valid. Thus

$$l_{1,1}^{2} = \overset{\max}{k\varepsilon 3} \left(-1, 0 + 7\right)$$

The whole $L^{(2)}$ is generated is this way as shown below.

$$\underset{L^{(1)}}{\begin{pmatrix} -1 & 0 & -1 \\ 7 & -1 & 3 \\ 3 & -1 & -1 \end{pmatrix}} \underset{L^{(1)}}{\begin{pmatrix} -1 & 0 & -1 \\ 7 & -1 & 3 \\ 3 & -1 & -1 \end{pmatrix}} \Rightarrow \underset{L^{(2)}}{\begin{pmatrix} 7 & -1 & 3 \\ 6 & 7 & -1 \\ -1 & 3 & -1 \end{pmatrix}}$$

While $L^{(2)}$ was computed using only $L^{(1)}$, the matrix $L^{(3)}$, is computed using both $L^{(1)}$ and $L^{(2)}$ as shown below, with the computation for each element given as

$$l_{i,j}^{3} = \overset{\max}{k\varepsilon K} \left(-1, l_{i,j}^{1} + l_{k,j}^{2}\right)$$

as before. This gives the computation of $L^{(3)}$ below:

$$\underset{L^{(1)}}{\begin{pmatrix} -1 & 0 & -1 \\ 7 & -1 & 3 \\ 3 & -1 & -1 \end{pmatrix}} \underset{L^{(2)}}{\begin{pmatrix} 7 & -1 & 3 \\ 6 & 7 & -1 \\ -1 & 3 & -1 \end{pmatrix}} \Rightarrow \underset{L^{(3)}}{\begin{pmatrix} 6 & 7 & -1 \\ 14 & 6 & 10 \\ 10 & -1 & 6 \end{pmatrix}}$$

Once the matrix $L^{(m)}$ is created, the iteration bound can be determined using Equation (8.6) as shown below. In this case, $m = 3$ as there are three delays, therefore $L^{(3)}$ represents the final iteration.

$$T_{\infty} = \overset{\max}{i, m\varepsilon 1, 2, \ldots, D} \left\{ \frac{l_{l,l}^{m}}{m} \right\} \tag{8.6}$$

For this example, this gives the following:

$$T_{\infty} = \left\{ \frac{7}{2}, \frac{7}{2}, \frac{6}{3}, \frac{6}{3}, \frac{6}{3} \right\} = 4$$

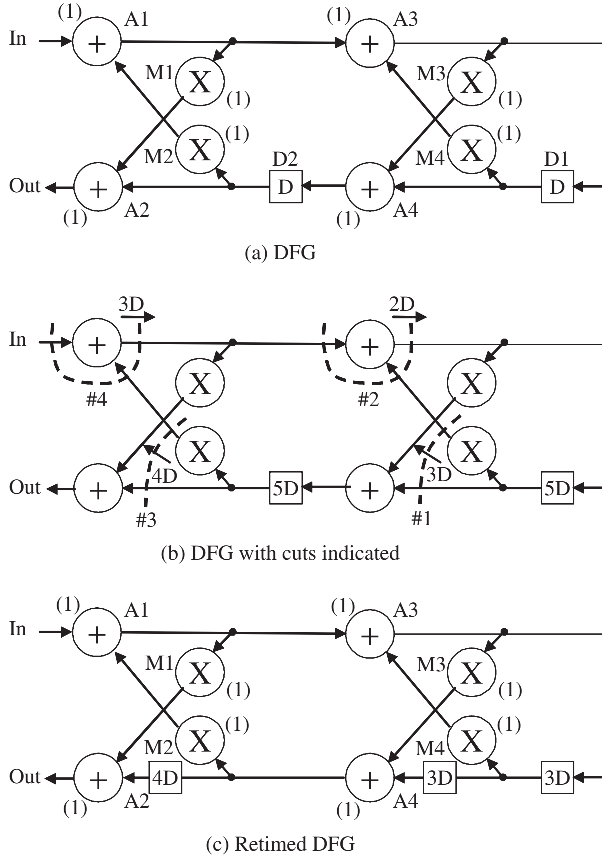


Figure 8.18 Lattice filter (Parhi 1999)

Example 2. Consider the lattice filter DFG structure given in Figure 8.18(a). Once again, a pipelined version has been chosen by selecting a single delay (1) for each processor.

The four possible matrix values are determined as follows:

D1 → M3 → A3 → D1
D1 → A4 → D2 and D1 → M4 → A3 → M3 → A4 → D2
D2 → M2 → A1 → A3 → D1
D2 → M2 → A1 → A3 → M2 → A4 → D2

thereby giving:

$$\begin{pmatrix} 2 & 4 \\ 3 & 5 \end{pmatrix}$$

The higher-order matrix L^2 is then calculated as shown below:

$$\underset{L^{(1)}}{\begin{pmatrix} 2 & 4 \\ 3 & 5 \end{pmatrix}} \underset{L^{(1)}}{\begin{pmatrix} 2 & 4 \\ 3 & 5 \end{pmatrix}} \Rightarrow \underset{L^{(2)}}{\begin{pmatrix} 7 & 9 \\ 8 & 10 \end{pmatrix}}$$

This gives the iteration bound as follows:

$$T_\infty = \underset{i, m \varepsilon 1, 2}{\max} \left\{ \frac{l_{i,i}^m}{m} \right\} \tag{8.7}$$

which resolves to:

$$T_\infty = \left\{ \frac{2}{1}, \frac{5}{1}, \frac{7}{2}, \frac{10}{2} \right\} = 5$$

Applying this scaling factor to the lattice filter DFG structure of Figure 8.18(b) gives the final structure of Figure 8.18(c), which has pipelined processors as indicated by the (1) expression added to each processor. This final circuit was created by applying delays across the various cuts and applying retiming at the processor level to transfer delays from input to output.

8.5 Parallel Operation

The previous section has highlighted methods to allow levels of pipelining to be applied to an existing DFG representation, mostly based on applying processor-level pipelining as this represents the greatest level applicable in FPGA realizations. This works on the principle that increased speed is required, as demonstrated by the results in Table 8.4, and more clearly speed improvements with FIR filter implementations. Another way to improve performance is to parallelize up the hardware (Figure 8.19). This is done by converting the SISO system such as that in Figure 8.19(a) into a MIMO system such as that illustrated in Figure 8.19(b).

This is considered for the simple FIR filter given earlier. Consider the 4-tap delay line filter given next

$$y(n) = a_0 x(n) + a_1 x(n-1) + a_2 x(n-2) + a_3 x(n-3) \tag{8.8}$$

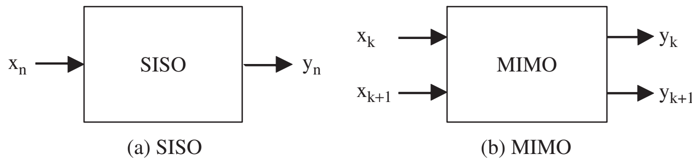


Figure 8.19 Manipulation of parallelism

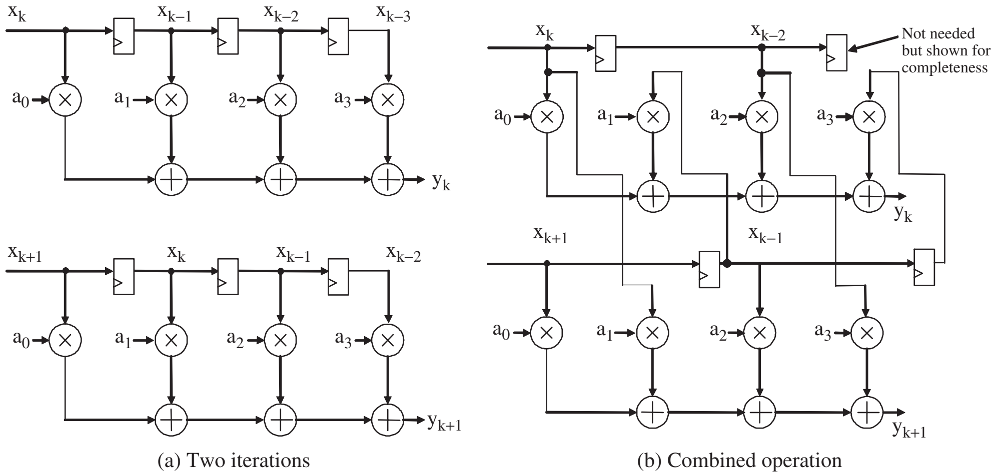


Figure 8.20 Block FIR filter

Assuming blocks of two samples per clock cycle, we get the following iterations performed on one cycle.

$$y(k) = a_0x(k) + a_1x(k-1) + a_2x(k-2) + a_3x(k-3)$$
$$y(k+1) = a_0x(k+1) + a_1x(k) + a_2x(k-1) + a_3x(k-2)$$

In the expressions above, two inputs $x(k)$ and $x(k+1)$ are processed and corresponding outputs $y(k)$ and $y(k+1)$ produced at the same rate. The data is effectively being processed in blocks and so the process is known as *block processing*, where k is given as the *block size*. Block diagrams for the two cycles are given in Figure 8.20(a). Note that in these structures any delay is interpreted as being k delays as the data is fed at twice the clock rate. As the same data is required at different parts of the filter at the same time, this can be exploited to reduce some of the delay elements, resulting in the circuit of Figure 8.20(b).

The FIR filter has a critical path of $T_M + (N-1)T_A$ where N is the number of filter taps which determines the clock cycle. In the revised implementation however, two samples are being produced per cycle, thus throughput rate is $2/(T_M + (N-1)T_A)$. In this way, block size can be varied as required, but this results in increased hardware cost.

Parhi (Parhi 1999) introduced a technique where the computation could be reduced by reordering the computation as below.

$$y(k) = a_0x(k) + a_2x(k-2) + z^{-1}(a_1x(k+1) + a_3x(k-1))$$

By creating two tap filters, given as $y(1k) = a_0x(k) + a_2x(k-2)$ and $y(2k) = a_1x(k+1) + a_3x(k-1)$, we re-cast the expressions for $y(k)$ as below:

$$y(k) = y(1k) + z^{-1}(y(2(k+1)))$$

The expression for $y(k+1)$ is rewritten as below:

$$y(k+1) = (a_0 + a_1)(x(k+1) + x(k)) + (a_2 + a_3)(x(k-1) + x(k-2))$$
$$-a_0x(k) - a_1x(k+1) - a_2x(k-2) - a_3x(k-1)$$

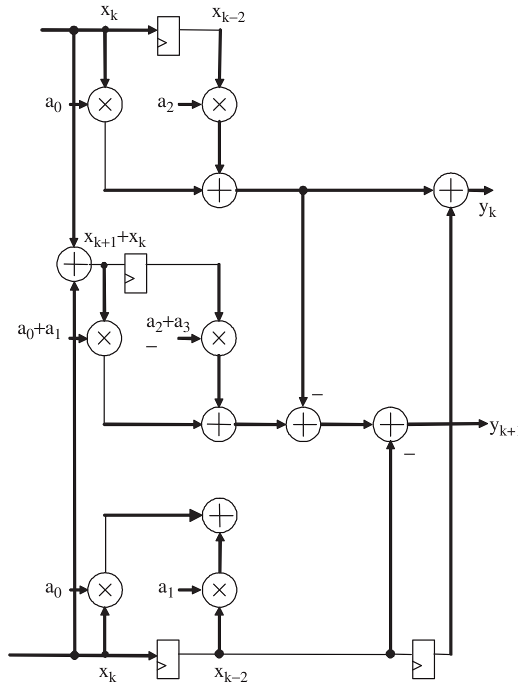


Figure 8.21 Reduced block-based FIR filter

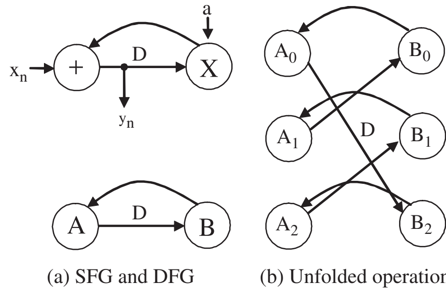
This results in a single 2-tap filter given below, comprising a structure with coefficients $(a_0 + a_1)$ and $(a_2 + a_3)$, thereby reducing the complexity of the original 4-tap filter. It does involve the subtraction of two terms, namely $y(k)$ and $y(2k + 1)$, but these were created earlier for the computation of $y(k)$. The impact is to reduce the overall multiplications by 2 at the expense of one addition/subtraction. This is probably not as important for an FPGA implementation where multiplication cost is comparable to addition for typical wordlengths. More importantly though, the top and bottom filters are reduced in length by 2 $(N/2)$ taps and an extra 2- $(N/2)$-tap filter is created to realize the first line in each expression. In general terms, filters have been halved, thus the critical path is given as $T_M + (N/2)T_A + 3T_A$ with three adders, one to compute $x(k) + x(k + 1)$, one to subtract $y(1k)$ and one to subtract $y(2(k + 1))$.

$$y(k+1) = (a_0 + a_1)(x(k+1) + x(k)) + (a_2 + a_3)(x(k-1) + x(k-2)) \\ - y(1k) - y(2(k+1))$$

8.6 Hardware Sharing

8.6.1 Unfolding

The previous section indicated how we could perform parallel computations in block. Strictly speaking this is known as *unfolding*. *Unfolding* is a transformation technique that can be applied



(a) SFG and DFG (b) Unfolded operation

Figure 8.22 Unfolded first-order recursion

to a DSP program to create a new program that performs more than one iteration of the original program. It is typically described using an unfolding factor typically J which describes the number of iterations by which it is unfolded. For example, consider unfolding the first-order IIR filter section, $y(n) = x(n) + by(n-1)$ by three, giving the expressions below:

$$y(k) = x(k) + by(k-1)$$

$$y(k+1) = x(k+1) + by(k)$$

$$y(k+2) = x(k+2) + by(k+1)$$

The SFG and DFG representation is given in Figure 8.22(a) where the adder is replaced by processor A and the multiplier by B. The unfolded version given in Figure 8.22(b) where A_0, A_1 and A_2 represent the hardware for computing the three additions and B_0, B_1 and B_2 represent the hardware for computing the three multiplications. With unlooped expressions, each delay is now equivalent to three clock cycles. For example, the previous value needed at processor B_0 is $y(n-1)$ which is generated by the delaying the output of A_2, namely $y(n+2)$, by an effective delay of 3. When compared with the original SFG, the delays would appear to have been redistributed between the various arcs for A_0–B_0, A_1–B_1 and A_2–B_2.

An algorithm for automatically performing unfolding was given in Parhi (Parhi 1999) and repeated here. It is based on the fact that the kth iteration of the node $U(i)$ in the J-unfolded DFG executes the $J(k+i)$th iteration of the node U in the original DFG.

Unfolding algorithm:

1. For each node U in the original DFG, draw the J nodes $U(0), U(1), \ldots, U(J-1)$.
2. For each edge $U \to V$ with w delays in the original DFG, draw the J edges $U(i) \to V(i+w)/J$ with $(i + w\%J)$ delays for $i = 0, 1, \ldots, J-1$ where % is the remainder.

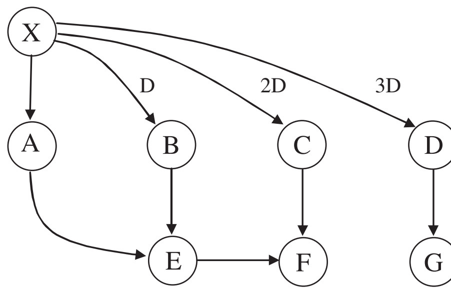
Consider the FIR filter DFG, a DFG representation of the FIR filter block diagram of Figure 8.23(a). Computations of the new edges in the transformed graphs, along with the computation of the various delays, is given below for each edge. This gives the unfolded DFG of Figure 8.23(b) which equates to the folded circuit given in Figure 8.23(a).

$$X0 \to A(0+0)\%2 = A(0), \text{ Delay}=\lfloor 0/2 \rfloor = 0$$
$$X1 \to A(1+0)\%2 = A(1), \text{ Delay}=\lfloor 1/2 \rfloor = 0$$
$$X0 \to B(0+1)\%2 = B(1), \text{ Delay}=\lfloor 1/2 \rfloor = 0$$
$$X1 \to B(1+1)\%2 = B(2), \text{ Delay}=\lfloor 2/2 \rfloor = 1$$
$$X0 \to C(0+2)\%2 = C(0), \text{ Delay}=\lfloor 2/2 \rfloor = 1$$

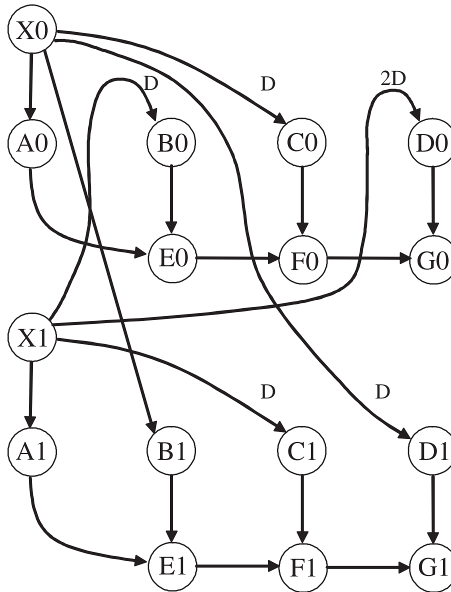
$$X1 \rightarrow C(1+2)\%2 = C(1), \text{ Delay}=\lfloor 3/2 \rfloor = 1$$
$$X0 \rightarrow D(0+3)\%2 = D(1), \text{ Delay}=\lfloor 3/2 \rfloor = 1$$
$$X1 \rightarrow D(1+3)\%2 = D(0), \text{ Delay}=\lfloor 4/2 \rfloor = 2$$

8.6.2 Folding

The previous section outlined a technique for a parallel implementation of the FIR filter structure. However in some cases, the desire is to perform hardware sharing or *folding* to reduce the amount of hardware by a factor, say k, and thus also reduce the sampling rate. Consider the FIR filter block diagram of Figure 8.24(a). By collapsing the filter structure onto itself four times i.e. folding by four, the circuit of Figure 8.24(b) is derived. In the revised circuit, the hardware requirements have

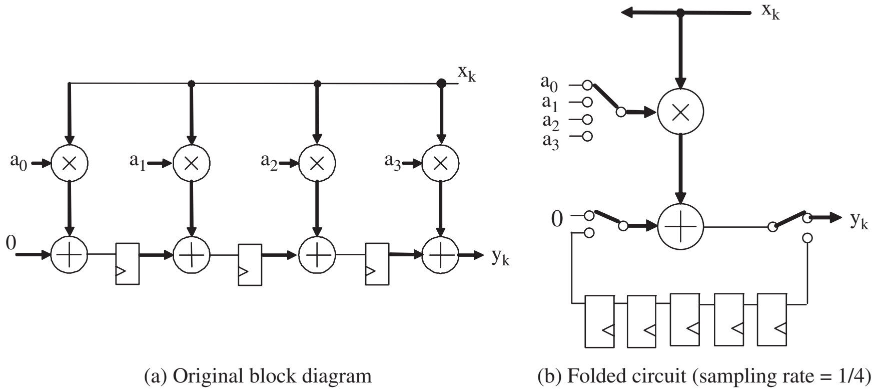


(a) SFG and DFG



(b) Unfolded operation

Figure 8.23 Unfolded FIR filter-block



(a) Original block diagram (b) Folded circuit (sampling rate = 1/4)

Figure 8.24 Folded FIR filter section

Table 8.5 Scheduling for Figure 8.24(b)

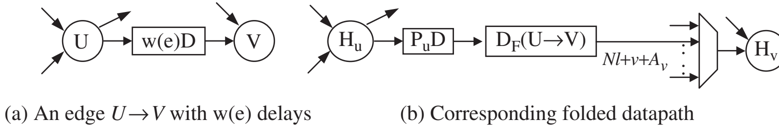
Cycle clock	Adder input	Adder input	Adder output	System Output
0	a_3	0	$a_3x(0)$	$(y(3)''')$
1	a_2	0	$a_2x(0)$	$(y(2)'')$
2	a_1	0	$a_1x(0)$	$(y(1)')$
3	a_0	0	$a_0x(0)$	$y(0)$
4	a_3	0	$a_3x(1)$	$(y(4)''')$
5	a_2	$a_2x(1)$	$a_2x(1) + a_3x(0)$	$(y(3)'')$
6	a_1	$a_1x(1)$	$a_1x(1) + a_2x(0)$	$(y(2)')$
7	a_0	$a_0x(1)$	$a_1x(1) + a_2x(0)$	$y(1)$
8	a_3	0	$a_3x(2)$	$(y(5)''')$
9	a_2	$a_2x(1) + a_3x(0)$	$a_2x(1) + a_2x(1) + a_3x(0)$	$(y(4)'')$

been reduced by four with the operation scheduled onto the single hardware units, as illustrated in Table 8.5.

The timing of the data in terms of the cycle number number is given by 0, 1, 2 and 3, respectively, which repeats every four cycles (strictly, this should by k, $k+1$, $k+2$ and $k+3$). It is clear from the table that a result is only generated once every four cycles, in this case on the 4th, 8th cycle, etc. The partial results are shown in brackets as they are not generated as an output. The expression $y(3)'''$ signifies the generation of the first part of $y(3)$, $y(3)''$, second part of $y(3)$ etc.

This folding equation (Parhi 1999) is given in Equation (8.9), where all inputs of a simplex component arrive at the same time and the pipelining levels from each input to an output are the same.

$$D_F(U \underset{\rightarrow}{e} V) = Nw(e) - P_u + v - u \tag{8.9}$$



(a) An edge $U \to V$ with w(e) delays (b) Corresponding folded datapath

Figure 8.25 Folding transformation

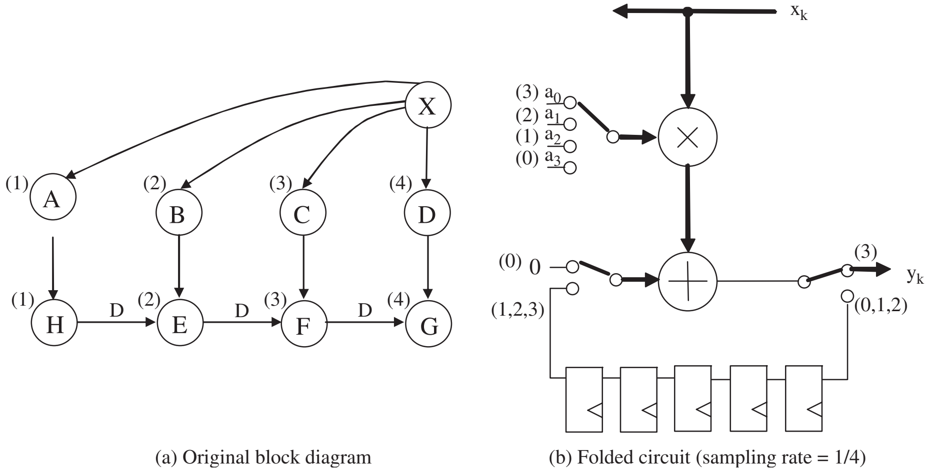
where $w(e)$ is the number of delays in the edge $U \underset{\rightarrow}{e} V$, N is the pipelining period, P_u is the pipelining stages of the H_u output pin and u and v are folding orders of the nodes U and V that satisfy $0 \le u, v \le N-1$. Consider the edge e connecting the nodes U and V with $w(e)$ delays shown in Figure 8.25(a), where the nodes U and V may be hierarchical blocks. Let the executions of the lth iteration of the nodes U and V be scheduled at time units $NL+u$ and $NL+v$ respectively, where u and v are folding orders of the nodes U and V that satisfy $0 \le$ u, v $\le$ N−1. The folding order of a node is the time partition to which the node is scheduled to execute in hardware (Parhi 1999). H_u and H_v are the functional units that execute the nodes U and V respectively. N is the folding factor and is defined as the number of operations folded onto a single functional unit. Consider the lth iteration of the node U. If the H_u output pin is pipelined by P_u stages, then the result of the node U is available at the time unit $Nl + u + P_u$, and is used by the $(l + w(e))$th iteration of the node V. If the minimum value of the data time format of H_u input pin is A_v, this input pin of the node V is executed at $N(l + w(e)) + v + A_v$. Therefore, the result must be stored for $D_F^{'}(U \underset{\rightarrow}{e} V) = [N(l + w(e)) + v + A_v] - [Nl + P_u + A_v + u]$ time units. The path from H_u to H_v needs $D_F^{'}(U \underset{\rightarrow}{e} V)$ delays, and data on this path are inputs H_v at $Nl + v + A_v$, as illustrated in Figure 8.25(b). Therefore, the folding equation for hierarchical complexity component is given in Equation (8.10).

$$D_F(U \underset{\rightarrow}{e} V) = Nw(e) - P_u + A_v + v - u \tag{8.10}$$

This expression can be systematically applied to the block diagram of Figure 8.24(a) to derive the circuit of Figure 8.24(b). For ease of demonstration, the DFG of Figure 8.26(a) is used. In the figure, an additional adder, H has been added for simplicity of folding. In Figure 8.26(a), we have used a number of brackets to indicate the desired ordering of the processing elements. Thus, the goal indicated is that we want to use one adder to implement the computations $a_3x(n)$, $a_2x(n)$, $a_1x(n)$ and $a_0x(n)$ in the order listed. Thus, these timings indicate the schedule order values u and v. The following computations are created as below, giving the delays and timings required as shown in Figure 8.26(a).

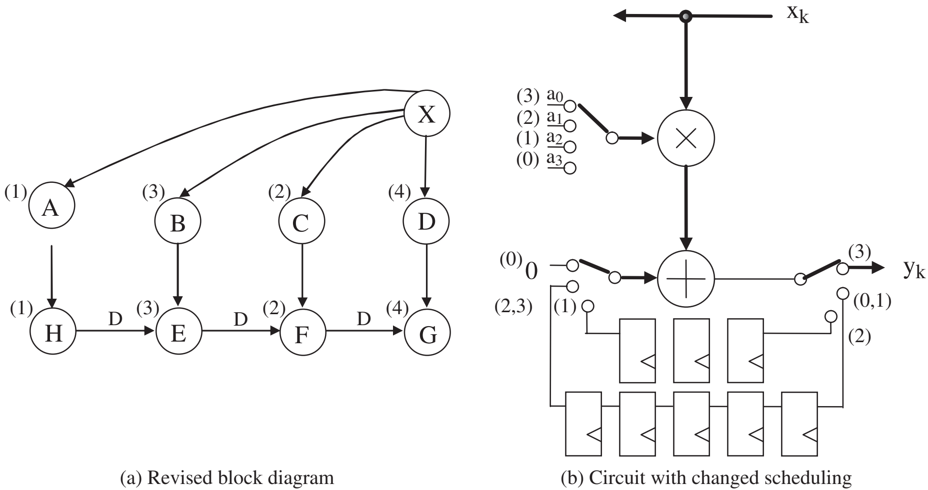
$$\begin{aligned}
D_{F(A\to H)} &= 4(0) - 0 + 1 - 1 = 0\\
D_{F(B\to E)} &= 4(0) - 0 + 2 - 2 = 0\\
D_{F(C\to F)} &= 4(0) - 0 + 3 - 3 = 0\\
D_{F(D\to G)} &= 4(0) - 0 + 4 - 4 = 0\\
D_{F(H\to E)} &= 4(1) - 0 + 2 - 1 = 5\\
D_{F(E\to F)} &= 4(1) - 0 + 3 - 2 = 5\\
D_{F(F\to G)} &= 4(1) - 0 + 4 - 3 = 5
\end{aligned}$$

Figure 8.27(a) shows how a reverse in the timing ordering leads to a slightly different folded circuit (Figure 8.27(b)) where the delays on the feedback loop have been changed and the timings on the multiplexers have also been altered accordingly. This example demonstrates the impact of changing the time ordering on the computation. The various timing calculations are shown below. The example below works on a set of order operations given as (1), (3), (2) and (4), respectively,



(a) Original block diagram (b) Folded circuit (sampling rate = 1/4)

Figure 8.26 Folding process



(a) Revised block diagram (b) Circuit with changed scheduling

Figure 8.27 Alternative folding

and requires two different connections between adder output and adder input with different delays namely 3 and 6.

$$
\begin{aligned}
D_{F(A\to H)} &= 4(0) - 0 + 1 - 1 = 0 \\
D_{F(B\to E)} &= 4(0) - 0 + 3 - 3 = 0 \\
D_{F(C\to F)} &= 4(0) - 0 + 2 - 2 = 0 \\
D_{F(D\to G)} &= 4(0) - 0 + 4 - 4 = 0
\end{aligned}
$$

$$D_{F(H \to E)} = 4(1) - 0 + 3 - 1 = 6$$
$$D_{F(E \to F)} = 4(1) - 0 + 2 - 3 = 3$$
$$D_{F(F \to G)} = 4(1) - 0 + 4 - 2 = 6$$

The application of the technique becomes more complex in recursive computations, as demonstrated using the second-order IIR filter example given in (Parhi 1999). In this example, the author demonstrates how the natural redundancy involved when a recursive computation is pipelined, can be exploited to allow hardware sharing to improve efficiency.

8.7 Application to FPGA

The chapter has briefly covered some techniques for mapping algorithmic descriptions in the form of DFGs, into circuit architectures. The initial material demonstrates how we could apply delay scaling to first introduce enough delays into the DFGs in order to allow retiming to be applied. This translates to FPGA implementations where the number of registers can be varied as required. As Chapter 5 had illustrated, use of pipelining is a powerful technique for producing high-performance FPGA implementations.

In the design examples presented, a pipelining of 1 was chosen as this represents the finest level of pipelining possible in FPGAs. This is done by ensuring *inter-iteration constraints* on the edges which can then be mapped into the nodes to represent pipelining. The delays remaining on the edges then represent the registers needed to ensure correct retiming of the DFG.

The chapter also reviews how to incorporate parallelism into the DFG representation which again is a realistic optimization to apply to FPGAs given the hardware resources available. In reality, a mixture of parallelism and pipelining is usually employed in order to allow the best implementation in terms of area and power that meets the throughput requirement.

8.8 Conclusions

The chapter has highlighted a number of techniques that allow the user to map an algorithmic DFG representation into a circuit architecture that is suitable for FPGA implementation. The technique concentrates on introducing registers into DFG representations and also transforming the DFG representation into a parallel implementation. These techniques are particularly suitable in generating IP core functionality for specific DSP functionality. As Chapter 11 illustrates, these techniques are now becoming mature, and the focus is moving to creating efficient system implementations from high-level descriptions where the node functionality may have been already captured in the form of IP cores. Thus, the rest of the book concentrates on this higher-level problem.

References

Leiserson CE and Saxe JB (1983) Optimizing synchronous circuitry by retiming *Proc. Third Caltech Conference on VLSI*, pp. 87–116.

Kung SY (1988) *VLSI Array Processors*. Prentice Hall, Englewood Cliffs, NJ.

Monteiro J, Devadas S and Ghosh A (1993) Retiming sequential circuits for low power. *Proc. IEEE Int. Conf. on Computer Aided Design*, pp. 398–402.

Parhi KK (1999) *VLSI digital signal processing systems : design and implementation*. John Wiley and Sons, Inc., New York.

9

The IRIS Behavioural Synthesis Tool

The descriptions in Chapter 7 have highlighted the range of tools and methodologies available for implementing FPGA-based DSP systems. In particular, the chapter highlighted the tools for synthesizing dedicated IP cores; however, the discussion in Chapter 8 highlighted some of the issues in creating the circuit architectures that are the basis of these IP cores. The exploration of the algorithmic concurrency in the form of levels of parallelism and pipelining in the algorithm representation typically a DFG, is instrumental in generating an efficient FPGA implementation. A number of simple examples were covered to illustrate the basic techniques, but the reality is that these techniques are difficult to explore on even reasonably complex algorithms. For this reason, there is a strong interest in developing synthesis tools to automate many of these techniques.

Behavioural synthesis tools accept a behavioural description of the functionality required of the hardware, then select suitable hardware building block components from libraries supplied by the tool vendor or the user, generate the interconnections needed among components, assign operations and data used in the behavioural description to the selected computational and storage components, determine the order in which operations execute on computational and storage components, and then generate a specification for a controller to implement this sequencing of operations. The resulting design can then be targeted to FPGA implementation using logic synthesis tools. To be effective, behavioural synthesis tools must accept user-specified constraints and/or optimization goals for the synthesized architecture. These typically include cost, clock cycle time, throughput, and latency. Such tools should also provide intuitive presentations of the results of the synthesis process that allow the user to quickly understand the synthesized architecture and the results of assignment and scheduling. The chapter describes the IRIS synthesis tool that has been developed at Queen's University, which captures many of these aspects and in effect, implement the techniques explored in Chapter 8. The tool represents one example of a number of behavioural synthesis, but gives details on implementing FPGA-based DSP systems from SFG descriptions. The chapter presents an expanded version of the paper by Yi and Woods (2006).

The chapter is organized as follows. Section 9.1 gives a brief introduction to behavioural synthesis tools with the aim of highlighting some of the processes described in the chapter. The IRIS synthesis tool is described in Section 9.2 which is based on the modular design procedure, described in Section 9.2.1. A key aspect of the synthesis flow is retiming and is described in Section 9.3. The examples in Chapter 8 were simple DFG descriptions, but the description in Section 9.4 shows the challenges in creating hierarchical implementations of SFG functionality. Section 9.5 then goes on to describe how hardware sharing can be implemented in hierarchically described functions.

FPGA-based Implementation of Signal Processing Systems R. Woods, J. McAllister, G. Lightbody and Y. Yi
© 2008 John Wiley & Sons, Ltd

9.1 Introduction of Behavioural Synthesis Tools

Some vendors provide CAD tools for digital system synthesis, typically using some form of synthesis, starting from the RTL level. A number of mature lower-level design tools for FPGA-based DSP design exist, such as Synplify (Synplify 2003) and Design Manager (Xilinx Inc. 2001). However, there is a clear need for higher-level synthesis tools to efficiently implement system and architectural level synthesis. At behavioural or algorithm level, the specification is given as an algorithm where basic structural elements are controllers and netlist. Architectural level synthesis (Vanhoof *et al*. 1993) generates a structural system description of the RTL level, starting from a behavioural (algorithmic) specification.

The following metrics are used to define a good synthesis system (Roy 1993).

Capability to synthesize complex designs: a synthesis system should be able to handle examples of reasonable size and complexity.
Shorter design cycles: a good synthesis system reduces the product life cycle and lowers chip cost significantly.
Extensive design space exploration: it is essential that a synthesis system explores a vast design space in an effort to produce optimal or near optimal designs.
Realistic constraints-driven synthesis: it should be able to work with realistic instead of abstract constraints.

However, some of the above metrics are conflicting and in a single processor implementation, an improvement in one metric tends to result in slippage in another. For example, extensive design space exploration will usually slow down the design process that can mean an increase in the design cycle. The various tasks associated with architectural level synthesis include scheduling of operators to control steps (scheduling), allocation of operators to operations in the description (allocation), assignment of operators to operations (binding) and allocation and assignment of memory modules to variables (allocation and binding).

Scheduling, resource allocation and binding are tasks involved in finding a satisfactory solution within the design space. Scheduling is an important sub-task in the synthesis process. Scheduling affects several aspects of the synthesized design, including the total number of functional units used, the total time of computation, the storage and interconnect requirements. The main focus here is to minimize the number of function units and therefore circuit area. Scheduling algorithms can be broadly classified as time-constrained or resource-constrained. In time-constrained scheduling, the number of function units are minimized for a fixed number of control steps. In resource-constrained scheduling, the number of control steps is minimized for a given design cost, i.e. the number of functional and storage units. Previous synthesis approaches include as soon as possible (ASAP) or as late as possible (ALAP) scheduling, which are the simplest ways to find a solution to a scheduling problem with precedence constraints (Dewilde *et al*. 1985). ASAP and ALAP scheduling have the disadvantage that nowhere does the algorithm refer to the resource usage. Therefore, no attempt is made to minimize the cost function and take into account resource constraints. List scheduling methods (Davidson *et al*. 1981) are targeted to resource-constrained problems by identifying the available time instants in increasing order and scheduling as many operations at a certain instant before moving to the next. The integer linear programming (ILP) method (Lee *et al*. 1989) tries to find an optimal schedule using a branch-and-bound search algorithm, which is relatively straightforward to automate. Most of these algorithms are based on a simplex method. The complexity of the ILP formulation increases rapidly with the number of control steps. In practice, the ILP approach is applicable only to very small-scale problems.

Force directed scheduling FDS (Paulin and Knight 1989) acts to minimize area hardware subject to a given time constraint by balancing the concurrency of operations, values to be stored, and data

transfers. The *iterative refinement* (IR) scheduling method (Park and Kyung 1991) is a heuristic scheduling algorithm that has the feature of escaping from a local minimum. Each edge describes a *precedence constraint* between two nodes, which is an *intra-iteration precedence constraint* if the edge has zero delays or an *inter-iteration precedence constraint* if the edge has one or more delays (Parhi 1999). Together, the *intra-* and *inter-iteration precedence constraints* specify the order in which the nodes in the signal flow graph (SFG) can be executed. If recursive nodes in noncritical recursive loops have loop flexibility (as will be illustrated later), then the nodes of that loop can be shifted to another time partition without violating *intra-* and *inter-iteration precedence constraints*. Exploration of the *inter-precedence constraints* generates better schedules and minimizes the number of allocated processors by using loop flexibility. All these algorithms described above are only concerned with the *intra-iteration precedence constraints*. Minnesota architecture synthesis (MARS, Wang and Parhi 1994) exploits both *inter-* and *intra-iteration precedence constraints*.

9.2 IRIS Behavioural Synthesis Tool

In the brief survey of design tools for mapping DSP systems to FPGAs in Chapter 7, it was shown that some of these efforts have focused on the design of FPGA-based architectural synthesis tools. However, there still are some limitations for each tool and there are clearly a number of important issues that are not adequately addressed by current systems. One example is that the computational delay generated by pipelining effects on system behaviour and system architecture. Although Handel-C, AccelChip (AccelFPGA 2002), JHDL (Bellows and Hutchings 1998) and MMAlpha (Derrien and Risset 2000) can pipeline the circuit using syntax, and System Generator (Xilinx Inc. 2000) can change the latencies of Xilinx processor blocks to pipeline the circuit, the resulting timing problems are left to the designer to solve. There is less work in the area of automatically implementing the circuit-level issues, such as data time format, pipelining level and numerical truncations, at the high level. In addition, many of the tools impose architectural constraints on the designer, limiting free exploration of a wide range of architectural alternatives. Specifically, the designer may want to exploit solutions with hardware sharing, but the circuits and corresponding control circuitry cannot be automatically generated by current tools. Finally, the tools have not allowed the architecture of the algorithmic design function to be determined in terms of specific technology features, and actually low-level design techniques such as pipelining.

This work described in this chapter, acts to solve the implementation-level issues during architectural synthesis for FPGA-based DSP design. The work is based on the existing architectural tools, IRIS (Trainor *et al*. 1997), which was developed at Queen's University Belfast and originally targeted at VLSI. The corresponding synthesis methodology in the IRIS architectural synthesis tools is reviewed in following sections.

It is important to have synthesis tools which perform optimizations at each level of the design flow. In the mid 1990s, it was argued that a design tool was required that allowed users to quickly develop VLSI circuit architectures from SFG representations using user-generated processing units. For this reason, the IRIS architectural synthesis tool was developed (Trainor *et al*. 1997); it provided an effective path to lower-level silicon design tools. Unlike other approaches, the emphasis in IRIS was to allow architectural exploration of algorithms by using user-generated processing units in order to generate optimal architectures based on those units. This was based on the premise of using in-house-developed processor cores provided by large companies and so this allowed development of solutions based on their own processing cores. The main advantage of IRIS was that it offered the designer full freedom to investigate a wide range of architectures using *user-preferred* blocks. This has gathered increasing importance as the silicon IP cores have been developed thereby increasing the need for these user-preferred blocks. The tool was also coupled to conventional VHDL synthesis

tools, and has been used to produce practical and realistic designs as it fully considered the effects of chip-level engineering issues. Some design issues such as the organization of the data entering and leaving the various processors, number systems employed, levels of pipelining and the handling of numerical truncation can change the characteristics of the complete system, particularly timing and latency (McGovern 1993), and thereby alter system function. A synthesis methodology, termed the modular design procedure (MDP, Trainor 1995) was proposed to provide a bridge between high-level algorithms to architecture mapping techniques and lower-level design tools. This became the core component of IRIS.

9.2.1 Modular Design Procedure

The MDP provides a way of incorporating all necessary implementation-level criteria into the architectural derivation process. Two processor performance issues were considered: space–time data format and parameterization of processor latencies (Trainor 1995).

Space–Time Data Format at the Processor Inputs and Outputs

The data format, or 'time shape', of data entering or leaving a processor may be defined as the position in the time of the bits, or digits, of the data value relative to each other (McGovern 1993). Figure 9.1 shows some examples of typical data time format.

The common procedure of pipelining processing elements at a fine-grained level, i.e. within the processor module, leads to non-parallel output format, so a tool's ability to cope with these nonstandard formats was essential. This is probably not as relevant with modern FPGA structures where most adders and multipliers consume inputs and produce outputs in bit-parallel form, as indicated by Figure 9.1(a). The detailed structure of the particular processor, and particularly the placement of internal pipelining registers, determines the data time shape at each processor input and output. It is also necessary to maintain information on the processor time shapes to determine

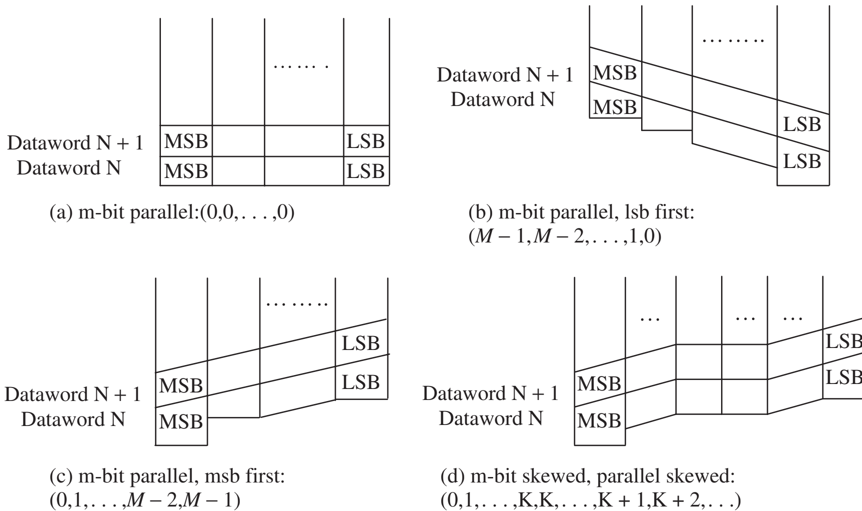


Figure 9.1 Typical data time formats

what extra circuitry, if any, needs to be placed between connected processors to convert between the formats of the data at the output of the first processor, and the format expected at the input of the second.

Parameterization of Processor Latencies

The latency in each datapath of processor must be incorporated into the processor model used during synthesis. The main reason is that the number of clock cycles required for a particular processor to produce its results has profound effects on the timing of data throughout the entire architecture. In order to illustrate the operation of the MDP, the example of the design of a simple Wallace tree multiplier architecture processor is considered and shown in Figure 9.2.

A Wallace tree multiplier given earlier in Figure 3.7 comprises CSAs connected to a fast adder. The obvious approach of pipelining is shown in Figure 9.2(b). In order to use the Wallace tree multiplier in conjunction with the MDP, the appropriate processor performance values, namely data time shapes and parameterized datapath latencies, must be determined and incorporated within models of the particular processor. An MDP example of partial product summation with Wallace trees, for 9-bit operands, is shown in Figure 9.2(c). The box represents the tree of CSA processors. The parameterized expressions within boxes along a particular datapath refer to the latency of that

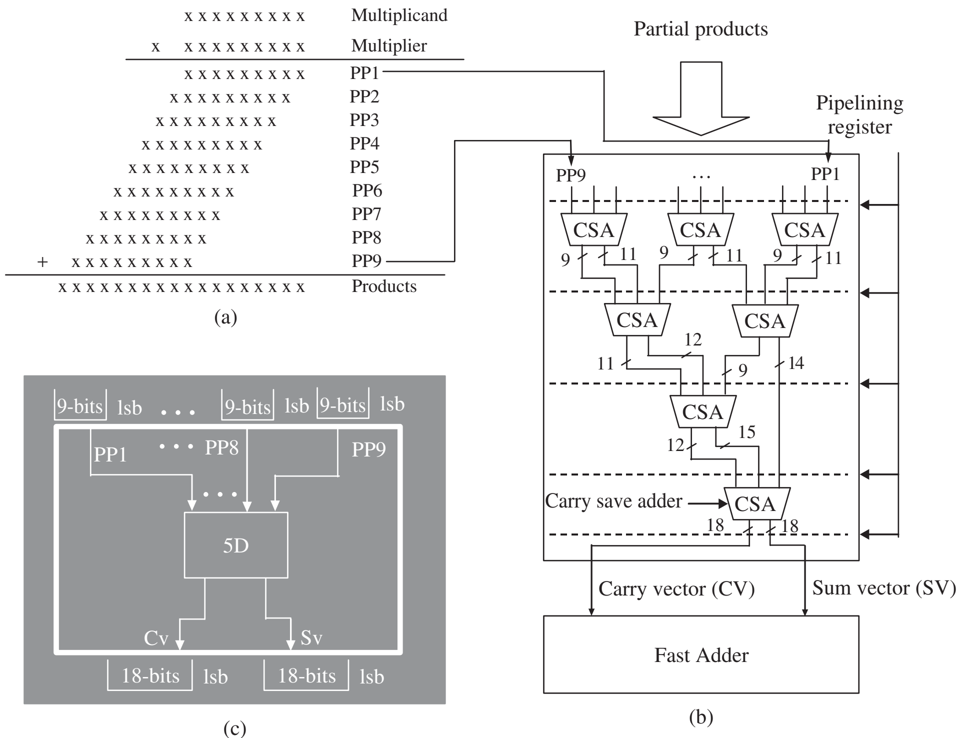


Figure 9.2 Wallace tree multiplier architecture compared with the MDP processor model

datapath, whilst the data time shape is shown graphically at the input and output of the model. The time shape is deduced by considering the timing of each bit of data as it passes through the multiplier. Notice that the construction of a processor model has abstracted away the detail, leaving only the latency and data time shape information. In addition to the word-length parameters and internal pipelining stages, some of the latency values in the processor model are also dependent on an additive truncation term t to reflect the fact that the latency through that datapath can be increased if numerical truncation is applied. The latency of the datapath is redefined as the time difference between when the first bit of the input enters the array and when the first usable bit emerges from the output. As was highlighted in Chapter 3, modeling of truncation in DSP applications is important, although it may seem odd to model truncation within the processor in this way, but this allows the impact of system level truncation decisions to be taken into account in the synthesis of systems as truncation can sometimes increase output latency. For example, if the output time shape is a skewed format and numerical truncation is needed, a certain number of the least significant bits are discarded and extra clock cycles may be needed to be taken into account before the first usable bit emerges. These extra cycles are reflected in the truncation term within the latency expression. The value of the term depends on the output data time shape and on which output bit the truncation is applied. The key feature of IRIS is the creation of the circuit architecture from the SFG representation which requires the use of a retiming process which is described next.

9.3 IRIS Retiming

The starting point for IRIS, is a SFG representation of the algorithm to be synthesized where each processing node in the SFG is then assigned a model of a fine-grained pipelined processor. With the processor MDP models now defined, these models can be embedded into the SFG representation. The architecture now requires to be rescheduled, i.e. retimed because of the changes in processor latencies. This retiming process is based on the retiming routine outlined via the cut-set theorem (Kung 1988, Leiserson and Saxe 1983) described earlier in Section 8.4.2; as with the cut-theorem, retiming routine in IRIS includes two steps, namely delay scaling and delay transfer. Delay scaling is used in a recursive architecture, i.e. one with feedback loops, in order to introduce sufficient delays around the feedback loop such that delay transfer can be carried out successfully. After the delay transfer routine, sufficient delays will have been created to appear on the appropriate SFG edges so that a number can be removed from the graph edges and incorporated into the processing blocks, in order to model the datapath latencies. This is referred to as embedding the processor models into the SFG (Trainor *et al*. 1997). The pure delays refer to any excess delays left on the graph edges, and are required for correct timing. This is the fundamental to the correct design of circuit architectures in IRIS.

When using the cut-set retiming procedure for a nonrecursive structures such as FIR filters, delay transfer is repeatedly applied to add or remove delays until sufficient delays appear within the architecture to allow processor embedding to take place. Recursive architectures, such as IIR filters, which exhibit feedback loops, cause additional problems. If delay transfer is applied to such loops, it cannot change the number of delays in the loops because delays that are removed from connections travelling in one direction must be transferred to the connections travelling in the opposite direction. Therefore, the delay transfer procedure is insufficient to carry out retiming successfully. The time scaling procedure can be applied by scaling up all the delay values by a factor α, called the *pipelining period*, and then redistributing this increased number of registers around the feedback loop, using the delay transfer procedure, in order to facilitate processor embedding. It is important to realize that applying time scaling to recursive architectures reduces the efficiency of such circuits by a factor of α, which may be defined as the ratio of sampling rate and clock rate.

Clearly, the problem is the determination of the optimal value of α for recursive structures. A value that is too small will lead to an incorrectly timed circuit, whilst too large a value will produce extra unnecessary registers in the architecture. This problem is solved by exploiting analysis carried out in Kung (1988) which was highlighted in Section 8.4.3 which identified the worst-case pipelining period (see Equations (8.3) and (8.4)).

9.3.1 Realization of Retiming Routine in IRIS

In order to carry out the retiming routines in IRIS, the SFG schematic is modelled as a doubly weighted graph (Christofides 1975), where the various external connectors and arithmetic processors represent the graph nodes and connections between these nodes represent the graph edges. For each edge, two weights can be defined, namely *SFGW* and *PW*. *SFGW*, previously called *SFGWeight* in earlier papers (Yi *et al*. 2005), refers to the number of delay elements on a particular edge, and *PW* (previously, *ProcWeight*) refers to the number of delay elements required on that edge for processor embedding to occur. The maximum allowable sampling period is α as defined in Equation (9.1).

In order to automate the procedure, IRIS employs the Floyd–Warshall algorithm (Parhi 1999) from graph theory, which determines the fastest loop in doubly weighted graphs. The algorithm solves the problem of finding a loop ϕ that satisfies Equation (9.2), where e is an edge in the weighted graph. Due to the fact that the algorithm has been cast as a minimization problem, the correct value of α is the reciprocal of the minimal value of Z(ϕ) (Trainor *et al*. 1997).

$$\alpha = \max \left\lceil \frac{\sum_{e \in \phi} ProcWeight(e)}{\sum_{e \in \phi} SFGWeight(e)} \right\rceil \tag{9.1}$$

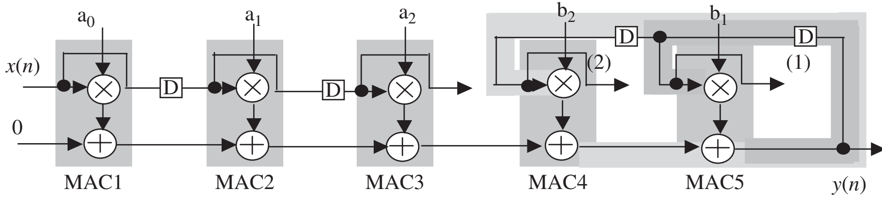
$$Z(\Phi) = \left\lceil \frac{\sum_{e \in \phi} SFGWeight(e)}{\sum_{e \in \phi} ProcWeight(e)} \right\rceil \tag{9.2}$$

When the pipelining period has been determined, all delay elements within the SFG are scaled up by this value, which is equivalent to adjusting the rate of the entire structure to its slowest loop.

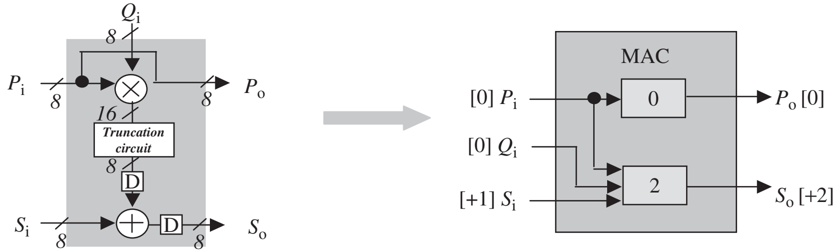
After delay scaling, the retiming procedure needs to be applied, in order to compensate for the different performance characteristics between the generic SFG processors and the practical models. The retiming routine, which has been solved as a linear programming problem in IRIS, minimizes the total number of delays used. A retiming r of a SFG G gives the nodes an integer value. The value $r(v)$ is the number of delays drawn from each of the incoming edges of node v and pushed to each of outgoing edges. The value $r(v)$ is positive if delays move from input to output, otherwise negative. Equation (9.3) represents the objective function for the linear programming problem, where $\beta(e)$ denotes the edge width, the symbol $e(v \longrightarrow ?)$ and $e(? \longrightarrow v)$ represent edges beginning and ending at node v respectively. The formulation of the linear constraints involves constructing expressions of the form of Equation (9.4), in which the edge e leaves node u and enters node v. The retiming functions within IRIS utilize the revised simplex technique (Gnizio 1985), which is widely used in automated linear programming solvers.

$$\min \sum_{v \in V} r(v) \left(\sum_{e(v \longrightarrow ?)} \beta(e) - \sum_{e(? \longrightarrow v)} \beta(e) \right) \tag{9.3}$$

$$r(u) - r(v) \geq \text{ProcWeight}(e) - SFG\text{Weight}(e) \tag{9.4}$$



(a) IIR filter implementtation using *zero* latency MACs



(b) Pipelined MAC processor and MDP model

Figure 9.3 Second-order IIR filter

In order to illustrate the process of retiming procedure in IRIS, the second-order IIR filter, given earlier as Equation (8.3), is used. The example was implemented in Section 8.4.3, but is given here again in order to illustrate how IRIS operates. A SFG is represented in Figure 9.3(a), where the MAC processor is assumed to have *zero* latency.

High-speed applications usually require pipelined designs, so all of the nodes of the circuit are to be implemented using pipelined MAC processors where adders and multipliers have one pipeline stage. Notice that in Figure 9.3(b), the 'width' for each connection in the structure equals the wordlength of the data travelling along that connection. Parameter values for the nodes shown in Figure 9.3(b) need to be supplied to calculate specific latency values and data time shapes for the filter design.

In this example, the MAC processor architecture and the corresponding MDP models are shown in Figure 9.3(b). The latency value of an output equals the maximum delays of all datapaths from all inputs to this output. For example, there are three datapaths from each input to output S_o in the pipelined MAC processor. The latency of output S_o is 2, which is equal to the delays from inputs P_i, Q_i to output S_o. The paths that have the maximum latency are used to define the input data time format at clock cycle 0. As all bits of the inputs or outputs in this model enter or leave the processor at the same time, the data time format is $[0, 0, 0 \cdots 0]$ labeled [0]. All of the other inputs and outputs data time formats will be referenced with respect to these paths. For example, the latency of the signal S_o equals 2 and is the maximum value of all output latencies. As the signal P_i and Q_i data time formats are equal to $[0, 0, 0, 0, 0, 0, 0, 0]$ shown in Figure 9.3(b) and the latency from S_i to S_o is 1, the signal S_i should enter the processor one clock after the signal P_i and Q_i, so the data time format is $[1, 1, 1, 1, 1, 1, 1, 1]$, labeled $[+1]$, in the processor MDP model. All inputs and outputs data time formats and datapath latencies are given in Figure 9.3(b). The SFG of Figure 9.3(a) exhibits two loops, highlighted in Figure 9.3(a). Using the MDP model shown in

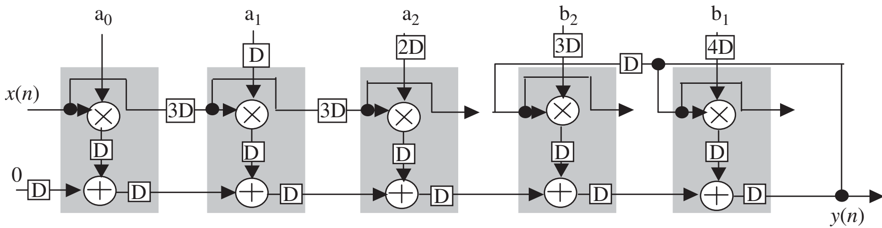


Figure 9.4 The second-order IIR filter implementation using pipelined MACs

Figure 9.3(b), evaluating Equation (9.1) gives the result that is displayed in Equation (9.5).

$$\alpha_1 = \left\lceil \frac{\sum_{e\in\phi} PW(e)}{\sum_{e\in\phi} SFGW(e)} \right\rceil = \frac{2}{1} = 2\{loop1\}$$

$$\alpha_2 = \left\lceil \frac{\sum_{e\in\phi} PW(e)}{\sum_{e\in\phi} SFGW(e)} \right\rceil = \frac{2+2}{1+1} = 2\{loop2\}$$

$$\alpha = \max(\alpha_1, \alpha_2) = 2 \tag{9.5}$$

Notice that the loop bound of loop 2 should equal to $(1 + 1 + 1)/(1 + 1) = 1.5$ which is different from α_2. The reason is that the data time format at the inputs may be at a different time, which is not considered in the delay scaling and delay transfer retiming procedure in IRIS. Given the connection between output $S_o(MAC4)$ and input $S_i(MAC5)$ in loop 2, the data time format of S_i is [+1], which means the input data enters *MAC5* one clock cycle later than the other inputs (P_i and Q_i). When the two delays are moved into the processor *MAC5* to incorporate the pipelined processor, an additional delay should be added in the edge of input $S_i(MAC5)$. This additional delay can be used to model the output latencies of MAC4 caused by pipelining. Pipelining period of the second-order IIR filter is 2 cycles, and hence all delay elements in the filter SFG must be scaled by a factor of 2 before retiming takes place. After delay scaling and retiming, the revised SFG is given in Figure 9.4.

The figures shows the time at which the inputs need to enter the design by indication of the number of delays on the inputs. In reality, these delays would not need to be implemented in the circuit, but would represent the control needed for correct operation.

9.4 Hierarchical Design Methodology

The description in the previous section has indicated how the retiming techniques outlined in Chapter 8 have been automated within IRIS. The output of the tool flow is then an optimized circuit architecture which could be coded up in VHDL and implemented using FPGA place and route design tools. The examples described represent DSP functionality where the design is implemented as a *flat* SFG representation. Increasingly, DSP hardware design flows need to cope with hierarchy

involving complex components Typically, designs are constructed in a hierarchical fashion where sub-components are created and then used to build larger systems. This brings specific problems when applying retiming in the way proposed, as these sub-components cannot be treated simply as *black box* components as they comprise internal timing.

This section describes how hierarchy is incorporated within the design flow, and introduces the concept of *white box* hierarchical management which allows some changes to the internal architecture of the previously created cores. The challenges are illustrated using a wave digital elliptic (WDE) filter example (Lawson and Mirzai 1990) which could not be synthesized using the original IRIS tools; thus, it represents a good example to illustrate the timing issues for hierarchical circuits using the original IRIS tools. The *white box* methodology is then proposed and applied to synthesize hierarchical structures. A revision to MDP is also proposed to deal with the hierarchical automatic synthesis issues.

9.4.1 White Box Hierarchical Design Methodology

During the synthesis process, one must decide whether to keep the hierarchy of the design or flatten it (Xilinx Inc. 1999). Flattening of the design may produce a smaller or faster design from a logic perspective, but it creates other complications in the design flow. First of all, blindly flattening the entire design may produce a single block of logic, large enough to overwhelm the capacity of the synthesis tool, resulting in unmanageable run times, or a suboptimal netlist. In addition, it may result in a highly disorganized netlist since the regularity of the various subsystems has not been maintained. As higher density FPGAs are introduced, the advantages of hierarchical designs considerably outweigh any disadvantages.

The disadvantage of a hierarchical approach compared with a flattened method is that design mapping may give a solution that is not as optimal across hierarchical boundaries. Whilst this can cause inefficient device utilization and decreased design performance, the hierarchical approach allows efficient design partitioning, a mix of options for individual block and incremental design changes, more efficient management of the design flow, and reduces design and debugging time by exploiting reuse. To benefit from a hierarchical approach, effective strategies are required to partition the design, optimize the hierarchy, and fine-tune the hierarchical synthesis process.

The hierarchical design flow is synonymous with the concept of IP core reuse (Keating and Bricaud 1998), where designs can be pre-generated for a range of system parameters so that they can be widely applied (McCanny *et al*. 1997). This is explored in more detail in Chapters 10 and 12. Thus, the use of IP cores is forcing a hierarchical approach.

Two different hierarchical synthesis methodologies have been defined by Bringmann and Rosenstiel (1997), and are based on:

- *black-box reuse*, where previously synthesized systems as components with no access to internal structure;
- *white-box reuse*, where previously synthesized systems as components with the possibility of changing internal architecture for pipelining or retiming reasons.

Both these approaches are commensurate with the IP core design approach, where the system designer may have limited knowledge and control over the complex components. It is then possible to develop models giving datapath latency and input timing, and perform synthesis in a hierarchical manner which is synonymous with the approach proposed in IRIS. As will be demonstrated using the WDE filter example (Parhi 1999), this is counterproductive and can lead to very inefficient solutions.

White box hierarchical management is used in Synplify Pro synthesis tools. During the synthesis process, the tools dissolve as much of the design's hierarchy as possible to allow efficient

optimization of logic across hierarchical boundaries while maintaining fast run times (Drost 1999). Synplify Pro then rebuilds the hierarchy as closely as possible to the original, with the exception of any changes caused by optimizations that straddles the hierarchical boundaries. This gives a final netlist that will have the same hierarchy as the original source code, ensuring that hierarchical register names remain consistent, and that major blocks of logic remain grouped together. This method of handling hierarchical boundaries, combined with the architecture-specific mapping, creates an efficient and effective optimization engine. However, designers need to manually consider the circuit level issues in terms of the selection of components in Synplify Pro (such as the latency of selected component). Manual solutions of circuit-level issues reduce design space exploration at the algorithm level, and increase time-to-market.

This section presents a design methodology based on *white box reuse*, which allows changes to be made to the internal delays without affecting the overall architecture. Along with the development of a hierarchical approach, the automatic synthesis methods for circuit level issues need to be added to hierarchical design. In the following section, MDP model extraction from previously synthesized subsystems in hierarchical circuits will be introduced.

9.4.2 Automatic Implementation of Extracting Processor Models from Previously Synthesized Architecture

The hierarchical timing issues will be demonstrated using a simple fifth-order WDE filter circuit shown Figure 9.5(a) which has been taken from Parhi (1999). As briefly described in Chapter 2, WDF filters have excellent stability properties (even under nonlinear operating conditions resulting from overflow and roundoff effects), low-coefficient wordlength requirements, inherently good dynamic range, etc. (Gazsi 1985). They are particularly attractive due to their low sensitivity to coefficient quantization (Lawson and Mirzai 1990). The fifth-order WDE filter is constructed from module A, B, C and D subsystems which are given in Figure 9.5(b–e), and are composed of multiplication and addition operations.

A number of steps are proposed to cope with this design in a hierarchical fashion. First, the processor models of the basic arithmetic operations are formulated. To achieve a high-performance FPGA implementation, each multiplier and adder is assumed to have a pipeline register on their outputs although various levels of pipelining can be modelled in IRIS. This is commensurate with FPGA implementations where pipelining at the component level represents the finest level of pipelining possible. Next, the basic arithmetic processors are used to synthesize the SFGs for the module subsystems. Then the synthesized module architectures are used to derive equivalent IRIS processors. Finally, the processor models for the module A, B, C and D subsystems are employed to synthesize the SFG for the fifth-order WDE filter.

The major difficulty is that some modules, namely A and D, may include registers that can impact the process of determining the pipelining period. For example, the delay in the module D subsystem results in the wrong pipelining period in the top-level circuit. Second, techniques whereby the original IRIS can automatically extract processor models from previously synthesized architectures have not yet been implemented and so a new technique is needed. Finally, there are some instances where the hierarchical circuit cannot be synthesized, even if the pipelining period can be calculated correctly because retiming using the linear programming used by IRIS (Trainor 1995, Trainor *et al*. 1997) will have no solution. The alternative is to flatten the hierarchical SFG level until the circuit can be synthesized. These approaches will be discussed in detail. It is important to note that these timing problems are applicable to a wider range of tools than IRIS when investigating hierarchical designs.

In the hierarchical SFG, the pipelining period for each processor will have been determined in advance for the MDP model. Thus, the calculation of the pipelining period α (Parhi 1999) of the hierarchical SFG becomes a complex task in IRIS. To calculate the whole circuit pipelining period,

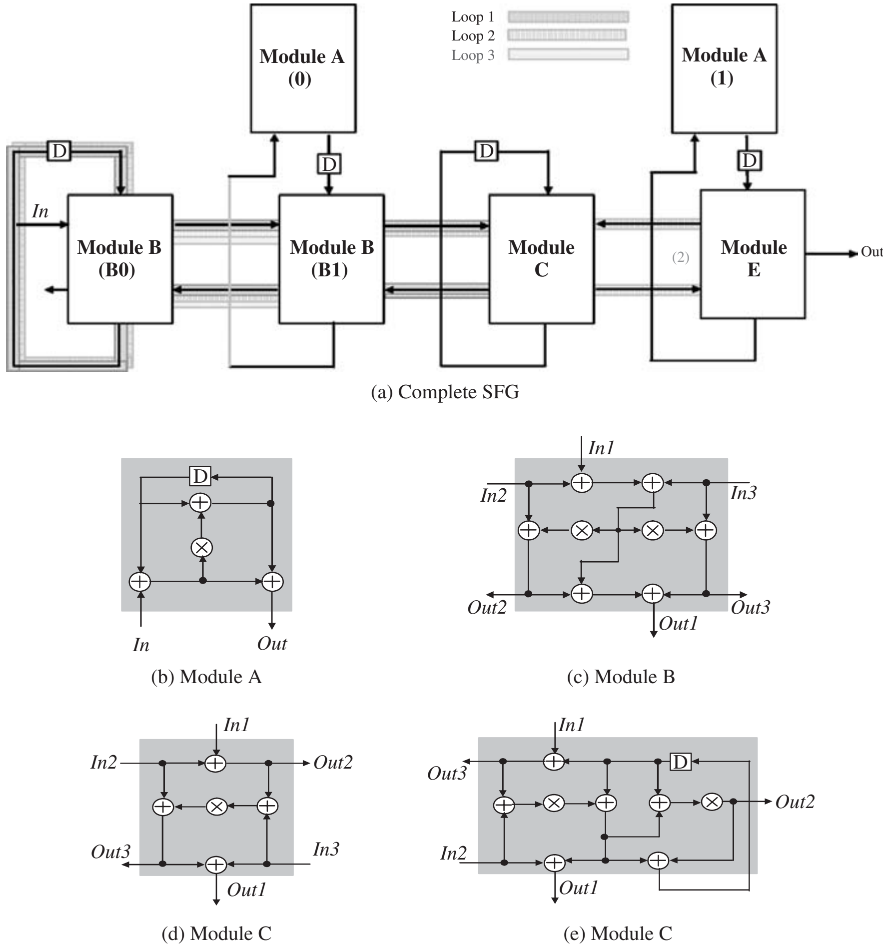
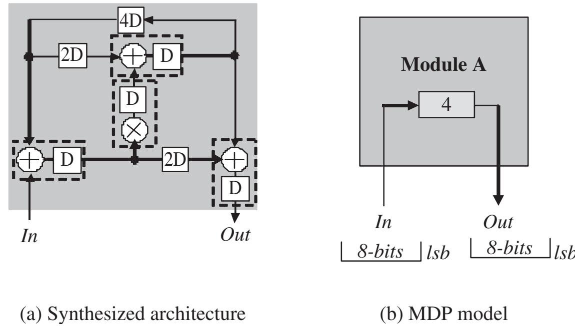


Figure 9.5 SFG of fifth-order WDE filter (pipelining of TA and TM is 0). Reproduced from *Hierarchical Synthesis of Complex DSP Functions Using IRIS* by Y. Yi & R. Woods, IEEE Trans on Computer Aided Design, Vol. 25, No. 5, © 2006 IEEE

it is necessary to assume that the pipelining period equals 1 and then allow IRIS to change it to the maximum pipelining period of all modules. When the IRIS synthesis and retiming routines are applied to the module level subsystems, which are modules A–D shown in Figure 9.5(b–e), a maximum pipelining period of seven clock cycles is calculated, which is the pipelining period of module D($\alpha_D = 7$) shown below. Thus, the whole circuit pipelining period changes to 7.

Module A: loop bound $\alpha_A = 3$
Module B: no loop
Module C: no loop
Module D: loop bound $\alpha_D = 7$

IRIS ensures that when the subsystem architecture is converted to a processor model, there are as few delays as possible contained within the subsystem model. As these delays cannot take part



(a) Synthesized architecture (b) MDP model

Figure 9.6 Module *A* subsystem (TM = 1 and TA = 1). Reproduced from *Hierarchical Synthesis of Complex DSP Functions Using IRIS* by Y. Yi & R. Woods, IEEE Trans on Computer Aided Design, Vol. 25, No. 5, © 2006 IEEE

in the retiming process initiated during the synthesis of the complete fifth-order WDE filter, this provides a better solution. Thus, any retiming latches that are necessary to change the input/output data flow in module A, will appear between the various processor instances. This allows IRIS to be more successful in minimizing the total number of retiming latches employed both within and between, the various subsystems in the fifth-order WDE filter.

The synthesized architecture for the module A subsystem is given in Figure 9.6(a). The retiming values of inputs and outputs can be obtained using a revised simplex linear programming (Vajda 1981). The retiming value of an input refers to the number of delays that move from outside to inside the subsystem and *vice versa* for the output. For example, the retiming value of input *In* is 4, which means that four delays are moved into subsystem. The retiming value of output *Out* is zero, which means that no delays are moved out from the subsystem.

In order to generate the MDP model for the retimed module A architecture shown in Figure 9.6(a), first consider the original SFG algorithm representation of module A subsystem shown in Figure 9.5(b), where the latency of multiplier and adder is 0. The input arrives and output leaves the subsystem at the same time, therefore the data time formats of the input and output are 0 and latency of output is also 0. After changing the latency of multiplier and adder to one, the IRIS retiming subroutine is used to deal with the timing problem caused by the latency of the multiplier and adder. The retiming value of the input *In* refers to the required number of delays when the latency of a processor changes, and is added to the input. The data time format of the output *Out* will be changed to 4 because the output *Out* appears after four clock cycles. The synthesized architecture is generated after moving four delays into module A and performing retiming. The implemented MDP representation of the module A subsystem is shown in Figure 9.6(b).

The module A subsystem with one input and one output, is a simple subsystem, and so the module B subsystem will be used to demonstrate the extraction method for multiple inputs and outputs. The module B subsystem has no loops with a pipelining period of 7 that would equal the system pipelining period. In the original module B, all the inputs/outputs appeared at the same time and their data time formats and output latencies were zero. After retiming, the various inputs and outputs of the module *B* subsystem may emerge during different clock cycles, as shown in Figure 9.7(a) where retiming values of inputs, e.g. *In1* and outputs are shown in brackets, e.g. (+5).

IRIS selects the maximum retiming value of all inputs in subsystem as β, and adds $[+\beta]$ delays to all inputs. The data time format of all outputs will be changed to β because β delays are inserted in the inputs. After IRIS retiming, the number of delays in the input edge will reduce the retiming value of that input, and the number of delays that equal the retiming value will be added to that

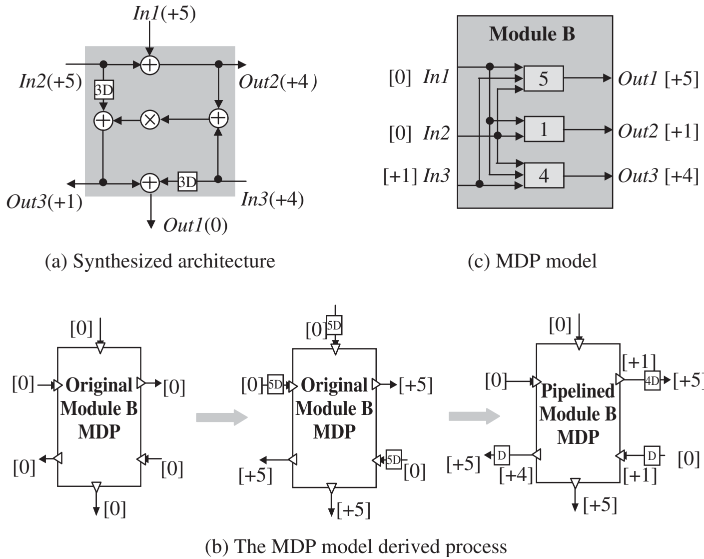


Figure 9.7 The module B subsystem (TM = 1 and TA = 1). Reproduced from *Hierarchical Synthesis of Complex DSP Functions Using IRIS* by Y. Yi & R. Woods, IEEE Trans on Computer Aided Design, Vol. 25, No. 5, © 2006 IEEE

output. For module B, the β value is 5, and these delays are added to all inputs. After retiming, the number of delays at the input *In3* edge is changed to 1 and equals the β value minus the retiming value of input *In3*. The number of delay at output *Out2* edge is the retiming value of *Out2* and is equal to 4. The whole procedure is illustrated in Figure 9.7(b). The latency of an output is defined as the time difference, in terms of clock cycles, between the input of the first bit of an operand and the emergence of the first bit of the corresponding output. For example, the output *Out1* is related to all inputs, the first input comes at clock cycle 0; the emergence time of the output is 5, therefore the latency of output *Out1* is 5. The resulting MDP model that will be used in next-level synthesis is shown in Figure 9.7(c).

9.4.3 Hierarchical Circuit Implementation in IRIS

Once processor models of the type described in the previous section have been obtained for all the different processing subsystems, the appropriate models are then embedded into the processing nodes of the higher level SFG. As previously stated, a rescheduling or retiming of the architecture is now required in the higher-level circuit because of the change in the lower-level processor latencies due to pipelined processor used.

Retiming of the hierarchical architecture is then carried out in two stages: delay scaling and processor embedding (Kung 1988). However, the previously used, double-weighted graph representation is not sufficient to solve the timing problem in hierarchical circuits because the inputs can emerge at different times. For example, the latency of output *Out3* in the module B subsystem is 5, therefore five delays are needed in the output *Out3* edge for embedding in the processor. The additional one delay will be added in the input *In3* edge after processor embedding. The original IRIS tool cannot represent the additional one delay in the weighted graph, and cannot therefore solve the problem efficiently. To address this, the treble-weighted graph is introduced where various external connectors, arithmetic processors and wire joints that permit branching of signal paths

represent the graph nodes, and the connections between these nodes represent graph edges. For each edge, three weights can be defined, as shown below. Whilst SFGW and PW were defined for the original IRIS, *PWI* (or strictly *ProcWeightIn*) is new and is added to solve the problem outlined above, with *PW* changed to *PWO*, i.e. *ProcWeightOut*.

SFGW – number of delays on a particular edge
PWO – number of delays required for processor embedding to occur
PWI – number of delays added to the edge for processor embedding

The *PWO* value of edges is related to the output of the starting processor and is the latency of the output that is the smallest value of data time format representation of that output pin (this is to cater for cases not discussed here, where data is bit skewed and not bit parallel). The *PWI* value of edges is associated with the input signal of the end processor, and equals the smallest value of data time format representation of that input pin. The *PWO* and *PWI* values of edges are related with connectors and wire joints and equal 0. The pipelining period α is changed to Equation (9.6) for the hierarchical circuit and the objective function and constraints for the linear programming problem is shown in Equation (9.7). The new *PWI* parameter is added to correct Equation (9.1) and Equation (9.4) because of hierarchical synthesis.

$$\alpha = \max \left\lceil \frac{\sum_{e\in\phi} PWO(e) - \sum_{e\in\phi} PWI(e)}{\sum_{e\in\phi} SFGW(e)} \right\rceil \tag{9.6}$$

$$\min \sum_{v\in V} r(v) \left(\sum_{e(v\longrightarrow ?)} \beta(e) - \sum_{e(?\longrightarrow v)} \beta(e) \right)$$
$$r(u) - r(v) \geq PWO(e) - SFGW(e) - PWI(e) \tag{9.7}$$

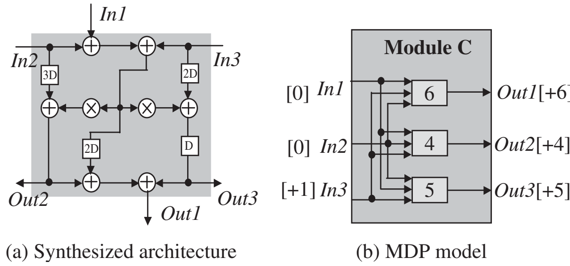
As the algorithm delays of the subsystems in low-level schematics can determine the pipelining period of the higher-level circuit, the calculation of the pipelining period, (α) of the hierarchical SFG, using Equation (9.6), becomes a complex task in IRIS. After the generation of the MDP models of subsystems, IRIS will calculate the whole fifth-order WDE filter pipelining period as outlined in the following subsections.

9.4.4 Calculation of Pipelining Period in Hierarchical Circuits

In the example of a fifth-order WDE filter, modules A and D include registers that may affect the pipelining period determined by the IRIS retiming subroutine. From the previous section, the pipelining period of the module level subsystems is 7, which is greater than the initial value 1. Therefore, the whole system pipelining period changes to 7. The synthesized MDP models for module A and B subsystems have been shown in Figures 9.6(b) and 9.7(c). The same automatic processor model extraction is applied to the modules C and D, and the synthesized circuits and graphical representations are shown in Figures 9.8 and 9.9.

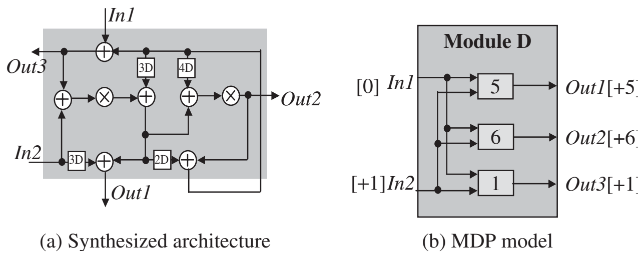
Once all of the processor models have been obtained, the appropriate models are then embedded into the processing nodes of the higher-level SFG. As previously stated, a retiming of the architecture is now required in the higher-level circuit because of the changes in the lower-level processor latencies.

Due to the fact that the delays in the subsystem of the original SFG cannot be used as *SFGW* in Equation (9.6) in the higher-level hierarchical synthesis, the pipelining period of hierarchical architectures using Equation (9.6) will produce the wrong result in many cases. For example, the



(a) Synthesized architecture (b) MDP model

Figure 9.8 Module C subsystem (TM = 1 and TA = 1)



(a) Synthesized architecture (b) MDP model

Figure 9.9 Module *D* subsystem (TM = 1 and TA = 1)

fifth-order WDE filter (Figure 9.5(a)) exhibits two recursive loops highlighted below. Using the *PWO*, *PWI* and *SFGW* values, evaluating Equation (9.6) gives the results below.

Loop1: B(0) ⟶ B(1) ⟶ C ⟶ B(1) ⟶ B(0) ⟶ D

$$\alpha_1 = \frac{\sum\limits_{e\in\phi} ProcWeightOut(e) - \sum\limits_{e\in\phi} ProcWeightIn(e)}{\sum\limits_{e\in\phi} SFGWeight(e)}$$
$$= \frac{1+1+4+4+5-0-0-1-1-0}{1} = 13$$

Loop2: Module B(0) ⟶ Module B(1) ⟶ Module C ⟶ Module D ⟶ Module C ⟶ Module B(1) ⟶ Module B(0)⟶ D

$$\alpha_2 = \frac{\sum\limits_{e\in\phi} ProcWeightOut(e) - \sum\limits_{e\in\phi} ProcWeightIn(e)}{\sum\limits_{e\in\phi} SFGWeight(e)}$$
$$= \frac{1+1+5+1+4+4+5-0-0-1-1-1-1-0}{1} = 17$$

The critical loop is *loop 2* whose pipelining period is 17, but the *loop 2* datapath includes one algorithm delay in module D subsystem and its pipelining period should have been 12, i.e. $\lceil 23/(1+1) \rceil$. In addition, the pipelining period is equal to infinity when Equation (9.6) is used to calculate *loop 3* shown in Figure 9.5(a). This is because the output *Out2* of module B subsystem is only related to the inputs *In1* and *In2*. The connections between module B(0) and module B(1) cannot form a loop. In order to solve these two problems, some other parameter needs to be added to the output in addition to data time format and latency.

Table 9.1 Relationship inputs vectors of module A–D

Output	Module A	Module B	Module C	Module D
Out 1	[In 1]	[In 1, In 2, In 3]	[In 1, In 2, In 3]	[In 1, In 2]
Out 2		[In 1, In 2]	[In 1, In 2, In 3]	[In 1, In 2]
Out 3		[In 1, In 2, In 3]	[In 1, In 2, In 3]	[In 1, In 2]

The *relationship inputs vector* of an output is introduced and describes inputs that have a datapath link to that output. For example, the relationship inputs vector of the output *Out2* in module *B* is [*In1*, *In2*] and means it is dependent on *In1* and *In2*. The relationship inputs vectors of Module A–D were shown in Table 9.1. If there is a datapath from input u to output v in a subsystem, the information (*latency*, *function delay*), called *datapath delay pair*, will be calculated. The *function delay* does not increase the latency, but generates the correct function and can be used as *SFGW*. The *latency* refers to the pipelining delay and is used as *PWO*. If there are multiple datapaths between u and v, the values of function delay are different, and IRIS needs to keep all the information pairs to calculate the pipelining period. If the function delay in the datapath delay pair is the same, the biggest latency value will be selected to calculate the pipelining period. The datapath delay pair is not needed if the subsystems do not include the function delay as the MDP model treble weights can be used to calculate the pipelining period. If the subsystems include the function delay, the MDP model and datapath delay pair will be used together to calculate the pipelining period. The MDP model determines the *PWI* and the datapath delay pair determines the *PWO* and *SFGW* values. The datapath delay pairs of module A and D, which include the function delay, have been shown in Table 9.2. The MDP model now includes four parameters that are *data time format*, *datapath latency*, *relationship inputs vector* and *datapath delay pair*.

By applying this method, the pipelining period of the higher-level circuit can be calculated. IRIS examines the SFG for loops and determines whether delay scaling is necessary. There are 25 loops in the top-level fifth-order WDE filter. The calculation of critical loops is shown below, and the pipelining period of the circuit is 13.

Loop1: Module B(0) $\longrightarrow$ Module B(1) $\longrightarrow$ Module C $\longrightarrow$ Module B(1) $\longrightarrow$ Module B(0) $\longrightarrow$ D

$$\alpha_1 = \frac{\sum_{e\in\phi} PWO(e) - \sum_{e\in\phi} PWI(e)}{\sum_{e\in\phi} SFGW(e)}$$
$$= \frac{1+1+4+4+5-0-0-1-1-0}{1} = 13$$

Table 9.2 Datapath delay pair of modules A and D

	Module A	Module D	
	In 1	In 1	In 2
Out 1	(4,0) (5,1)	(5,0) (12,1)	(4,0) (11,1)
Out 2		(6,0) (13,1)	(5,0) (12,1)
Out 3		(1,0) (8,1)	(7,1)

Loop2: Module B(1) $\longrightarrow$ Module C $\longrightarrow$ Module B(1) $\longrightarrow$ Module A $\longrightarrow$ D

$$\alpha_2 = \frac{\sum\limits_{e\in\phi} PWO(e) - \sum\limits_{e\in\phi} PWI(e)}{\sum\limits_{e\in\phi} SFGW(e)}$$

$$= \frac{1+4+5+4-0-1-0-0}{1} = 13$$

9.4.5 *Retiming Technique in Hierarchical Circuits*

After delay scaling, the linear programming routine is employed to carry out processor embedding. In some situations, the hierarchical circuit cannot be synthesized, even when the pipelining period can be calculated correctly because the linear programming has no solution. The retiming routine of IRIS will carry out synthesis from the top level and generate the circuit. Otherwise, it will flatten the conflicting subsystem and synthesize the circuit again. If there is no result for circuit synthesis exists in this level, the IRIS tool will continue iteratively to flatten the conflicting subsystem and retime until the circuit can be synthesized. Currently, the IRIS tools cannot do this automatically.

The linear programming can have no solution because some constraints are in conflict. For example, the constraint of edge from output *Out2* of module B(0) to input *In2* of module B(1) is $r(\mathrm{B0}) - r(\mathrm{B1}) \geq 1$, using Equation (9.7), and the constraint of edge from output *Out3* of module $B(1)$ to input *In3* of module B(0) is $r(\mathrm{B1}) - r(\mathrm{B0}) \geq 3$. These two constraints are in conflict, so the linear program will have no solution. In this case, the IRIS retiming routine flattens modules B(0), B(1), C and D, performs retiming, and translates the flattened circuit back to the hierarchical circuit. The treble-weighted graph for the partly flattened SFG of the fifth-order WDE filter is given in Figure 9.10, where the inputs and outputs of the flattened subsystems are used as graph nodes with zero values for *PWO* and *PWI*. After retiming, the synthesized architecture for the fifth-order WDE filter is as shown in Figure 9.11. It can be translated back to a hierarchical representation. For example, the module B(0) subsystem is flattened in the retiming procedure and the inputs and outputs of the module B(0) are marked and related with the corresponding nodes of the treble-weighed graph in order to be translated back to the module B(0) subsystem after retiming. In order to test methodology correctness, the completely flattened circuit of the fifth-order WDE filter is built up and synthesized using the IRIS tools. The retimed SFG is the same as in Figure 9.11.

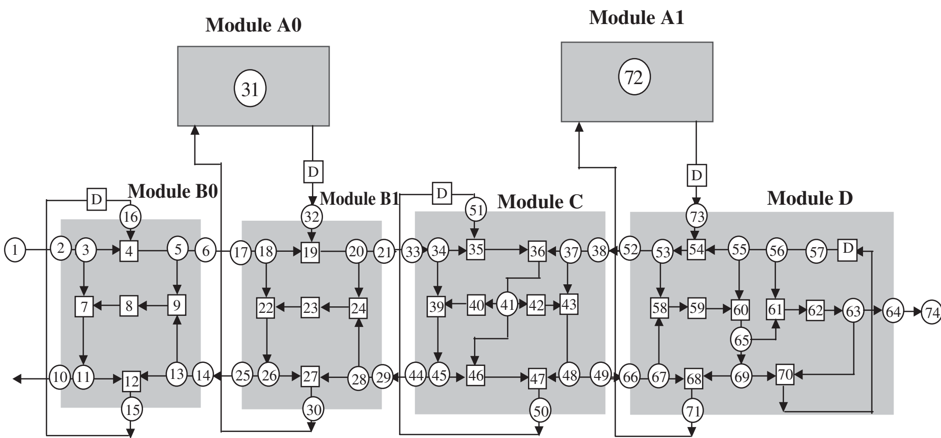


Figure 9.10 Treble-weighted graph for partly flattened SFG of the fifth-order WDE filter

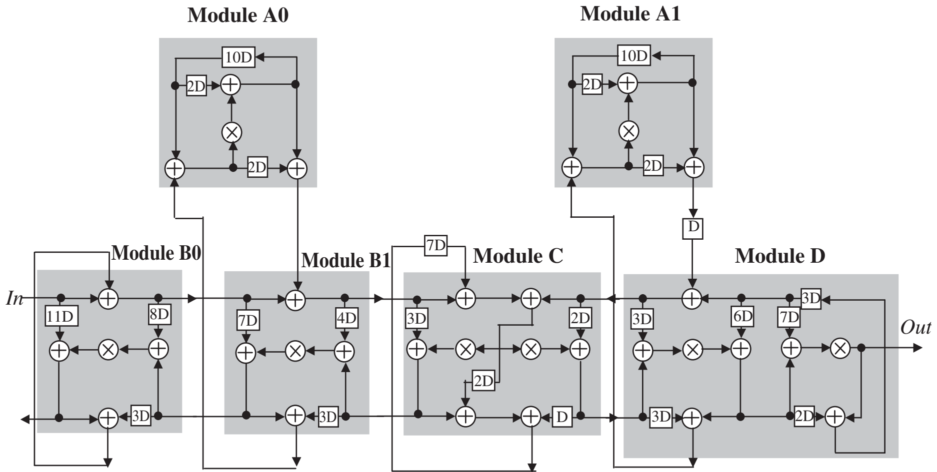


Figure 9.11 Retimed SFG of the fifth-order WDE filter pipelining level of TA = 1 and TM = 1

During synthesis, IRIS produces the structural VHDL description of the fifth-order WDE filter circuit. Each processor type (multiplier and adder) and delay blocks are defined and their interconnection specified. Separate blocks of code for the modules (A0, A1, B0, B1, C and D) and the filter architecture are generated. This permits the complete hierarchical VHDL description to be loaded into Synplify Pro for logic synthesis, and hence physical implementation via the Xilinx Design Manager.

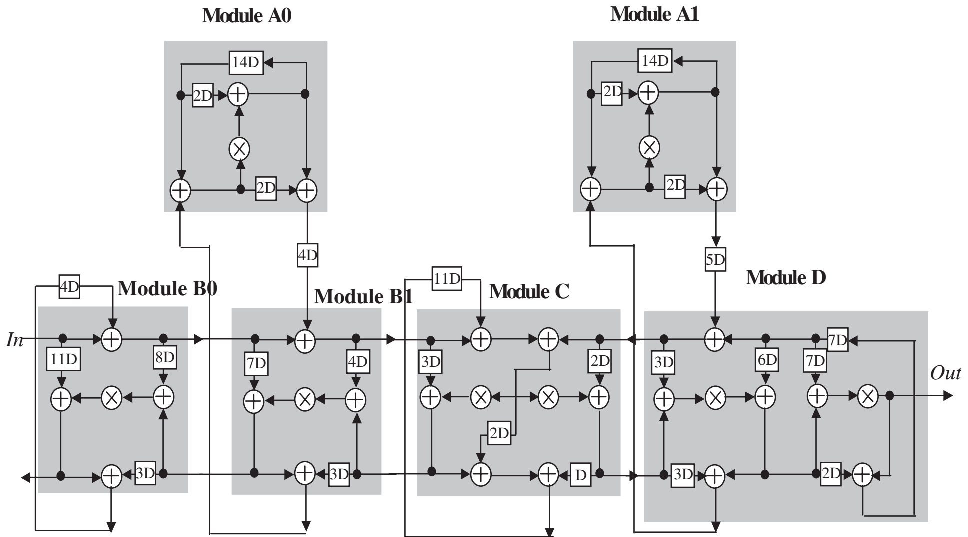


Figure 9.12 Retimed SFG of the fifth-order WDE filter using original IRIS tools (TA = 1 and TM = 1)

Table 9.3 Comparison of the characteristics

Architecture	Critical path	Hardware elements		
		Adder	Multiplier	Delay
WDEF	3TM + 11TA	26	8	7
PWDEF	TM	26	8	104
OPWDEF	TM	26	8	132

Table 9.4 Virtex-II post-layout results for sampling rate

Circuit name	Clock (MHz)	Critical paths (ns)		
		Logic	Route	Total
WDEF	25.6	24.2	14.9	39.0
PWDEF	173.0	3.8	1.9	5.8

Table 9.5 Virtex-II implementation (post-layout) results for area

Circuit name	Clock (MHz)	Area				
		LUTs	MULT	SRL16	FFs	Slice
WDEF	25.6	286	8		56	161
PWDEF	173.0	430	8	144	464	448

A comparison of the fifth-order WDE filter implementations in terms of critical paths and hardware overhead is given in Table 9.3. WDEF is the original filter shown in Figure 9.5 and PWDEF is the optimized design shown in Figure 9.11. The circuit shown in Figure 9.12 is the original pipelined WDE filter (OPWDEF) generated by the original IRIS tools. The PWDEF and OPWDEF circuits have a 93% critical path reduction over the WDEF circuit. PWDEF gives a 27% reduction in delays compared with OPWDEF. Tables 9.4 and 9.5 give details on 8-bit input/output coefficients post-layout information for WDEF and PWDEF system using a Virtex-II XC2V80fg144-5 chip. The implementation results obtained by the design flow show that the system performance can reach up to 173 MHz for the PWDEF circuit. The retimed designs have smaller routing interconnection delays than their non-pipelined counterpart because of the pipelining registers.

9.5 Hardware Sharing Implementation (Scheduling Algorithm) for IRIS

Section 9.4 presented the main functions of the new IRIS tool, which allow the designer to quickly and automatically synthesize a circuit architecture from a hierarchical algorithmic SFG representation, and evaluate the effects of the latency and data time format on the architectural synthesis. In regard to architectural exploration, the designer may want to investigate solutions with hardware sharing. This will require complex control, therefore a means of scheduling and generating control for the circuit is essential. This section describes the scheduling algorithm and controller adopted in the IRIS tools, namely the extended Minnesota architecture synthesis (EMARS) scheduling

algorithm for hierarchical scheduling (Yi *et al*. 2002). This is an extension to the MARS scheduling algorithm presented by (Wang and Parhi 1994) which exploits both *inter*- and *intra-iteration precedence constraints*. This section concentrates on adopting the EMARS scheduling algorithm for IRIS in order to realize scheduling and hardware sharing architectural transformations. A revised folding transformation technique provides a systematic technique for designing control circuits for hardware where several algorithm operations are time-multiplexed onto a single functional unit.

The main changes to the MARS algorithm are that the calculation of loop bound is changed to apply the complex component and hierarchical circuit. The corresponding changes are needed for initial scheduling, resolving conflict, scheduling and resource allocation. The folding transformation (Parhi 1999) provides a systematic technique for designing control circuits for hardware where several algorithm operations are time-multiplexed onto a single functional unit. The derivation of the folding equation is based on this technique. In the hierarchical architecture, the folding equation is changed to suit the hierarchical architecture. The main steps of EMARS algorithm are described as follows.

Step 1: Scheduling and Resource Allocation Algorithm for Recursive Nodes

The objective of high level architectural synthesis is to translate an algorithmic description to an efficient architectural design while using realistic technological constraints. The resultant architecture must maintain the original functionality while meeting speed and area requirements. As was previously demonstrated, the recursive sections of the system determine the maximum sampling rate. The scheduling for recursive nodes is considered first.

Step 1.1: Loop Search and Iteration Bound

When a loop is located, the original MARS calculates the loop bound as follows:

$$T_{LB_j} = \frac{T_{L_j}}{D_{L_j}} \tag{9.8}$$

where T_{L_j} = the loop computation time of loop j and D_{L_j} = the number of loop delays within loop j. The iteration bound can be calculated from the loop bound values by locating the maximum loop bound. As the original MARS only considers the simple components with all inputs arriving and outputs leaving at the same time, the computation equation of the complex and hierarchical architecture block is considered here. In EMARS, the loop bound and the computation time of critical path are calculated as follows:

$$T_{LB_j} = \frac{\sum_{e\in j} PWO(e) - \sum_{e\in j} PWI(e)}{\sum_{e\in j} SFGW(e)} \tag{9.9}$$

$$T_{\text{Crit}} = \sum_{e\in p} PWO(e) - \sum_{e\in p} PWI(e) \tag{9.10}$$

where the meanings of *PWO(e)*, *PWI(e)* and *SFGW(e)* are same as those given for Equation (9.6). The values j and p refer to loop j and datapath p respectively.

The second-order IIR filter is used to illustrate the limitations of the original MARS scheduling algorithm and to indicate that the new Equations (9.9), (9.10) are correct for the second-order IIR filter.

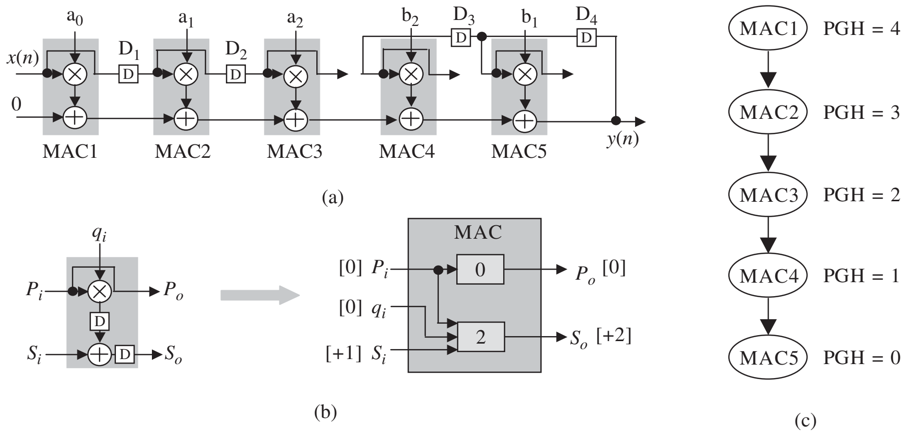


Figure 9.13 (a) SFG of the second order IIR filter using MACs; (b) pipelined MAC architecture; (c) dependence graph and PGH value for nodes. Reproduced from *Hierarchical Synthesis of Complex DSP Functions Using IRIS* by Y. Yi & R. Woods, IEEE Trans on Computer Aided Design, Vol. 25, No. 5, © 2006 IEEE

Consider the second-order IIR filter benchmark as shown in Figure 9.13(a), which consists of five MACs. It is assumed that each multiplier and adder has a computation time of 1 unit, which is commensurate with FPGA implementation. The MDP model of MAC is shown in Figure 9.13(b) and the dependence graph and *PGH* values for the nodes of the filter are shown in Figure 9.13(c). This filter contains a recursive section which has two recursive nodes (MAC4 and MAC5), and a non-recursive section which has three non-recursive nodes (MAC1, MAC2 and MAC3). There are 2 loops in this filter:

Loop1: MAC5(Pi)$\longrightarrow$MAC5(So)$\longrightarrow$ D_4
Loop bound $= \frac{2-0}{1} = 2$

Loop2: MAC4(Pi)$\longrightarrow$MAC4(So)$\longrightarrow$ MAC5(Si)$\longrightarrow$MAC5(So)$\longrightarrow$ $D_4 \longrightarrow$ D_3
Loop bound $= \frac{2-1+2-0}{1+1} = \frac{3}{2}$

Critical path:MAC1(So)$\longrightarrow$MAC2(So)$\longrightarrow$MAC3(So)$\longrightarrow$Mac4(So)$\longrightarrow$MAC5(So)
Tcrit $= 2 - 1 + 2 - 1 + 2 - 1 + 2 - 1 + 2 = 6$

Therefore the total number of loops required to schedule the recursive nodes is 2: *Loop1* and *Loop2*. In the filter, the iteration bound is 2 units and the computation time of the critical path is 6 units.

Step 1.2: Initial Schedule

The iteration bound (or pipelining period) has been calculated in step 1.1. The iteration time partition is defined to be the time step at which a task is executed modulo of the iteration bound (Wang and Parhi 1994). For example, the iteration bound of the second-order IIR filter is 2, therefore there are two iteration time partitions (0,1). If a task is scheduled at time step 5, then the task is assigned to

time partition 1 which is 5 modulo 2. The processor lower bound is given in MARS as follows:

$$(\text{Lower bound})_u = \left\lceil \frac{N_u \times T_u}{P_u \times T} \right\rceil \tag{9.11}$$

where N_u = number of U-type operations, T_u = computation time of a U-type operation, P_u = pipelining level of a U-type processor and T = iteration period.

Compared with the single operation in the MARS scheduling algorithm, the different outputs in the hierarchical architecture in IRIS may have different computation times and pipelining levels. In EMARS, T_u refers to the longest computation time of all outputs of a U-type operation and P_u is the highest pipelining level of a U-type operation. The T_u and P_u of the *MAC* are related to output S_o. The lower bound of the *MAC* in the second-order IIR filter is given in Equation (9.12), which means 3 MAC processors are required.

$$(\text{Lower bound})_{\text{MAC}} = \left\lceil \frac{5 \times 2}{2 \times 2} \right\rceil = 3 \tag{9.12}$$

At this point, the schedule matrices are created with one matrix for each operation type. The matrices represent each recursive node scheduled in one iteration time partition, where rows represent iteration time partitions and columns represent loops and loop sections. Each loop or loop section is assigned to a set of columns in such a way that the first set of columns corresponds to the most critical loop and the last set to the least critical loop. The most critical loop is firstly scheduled, starting from time 0 and maintaining the *intra-iteration precedence constraints*.

In IRIS, the *intra-dependence constraint* used to generate scheduling matrices is different from the simple operation because all inputs of the revised MDP model in IRIS come into the processor at different times. In the EMARS algorithm, the starting scheduling time of the node refers to the time of an input pin with data time format [0]. For example, the scheduling time of the MAC processor shown in Figure 9.13(b) is determined by the input pins P_i and Q_i. If there is an *intra-dependence constraint* from processor A to B, the scheduling time of the processor B can be calculated using the following equation:

$$T_B = T_A + L_A - D_B \tag{9.13}$$

where T_A = the scheduling time of processor A, T_B = the scheduling time of processor B, L_A = the latency of the output pin of processor A and D_B = the minimum value of data time format of the input pin of processor B.

For the second-order IIR filter example, *Loop1*, the critical loop, is scheduled first and the node MAC5 is scheduled at time 0. Because there is an *intra-dependence constraint* between MAC4 and MAC5 in *Loop2* and the starting time partition of MAC5 is 0, the scheduling time of node MAC4 is calculated as -1 using Equation (9.13). MAC4 is a wrapped node (Wang and Parhi 1994), i.e. a node which has been scheduled at time steps which are either negative (i.e. $t < 0$) or greater than or equal to T (i.e. $t \geq T$. Non-wrapped nodes are those which are scheduled at time steps equal to the time partitions (i.e. $0 \leq t \leq T - 1$). Wrapped nodes are identified in the schedule with a superscript of +l or -1. The schedule matrix and initial schedule are shown in Tables 9.6 and 9.7, where L1 and L2 refer to the loops, and M1, M2 and M3 are the physical MAC processors.

The loop flexibility available for each member of set L defines the number of iterations that nodes of a loop may be shifted before violating the inter-iteration precedence constraint. MARS calculates the loop flexibility F for each loop of L by the following equation, where T is iteration

Table 9.6 Schedule matrix for 2nd-order IIR filter

Time	L1	L2
0	MAC5	
1		MAC4^{+1}

Table 9.7 Initial schedule for 2nd-order IIR filter

Time	M1	M2	M3
0	MAC5		
1	MAC4^{+1}		

period, D_{L} is number of loop delays and T_{L} is loop computation time.

$$F = T \times D_{\mathrm{L}} - T_{\mathrm{L}} \tag{9.14}$$

For EMARS, $T_{\mathrm{L}} = \sum_{e \in \mathrm{L}} PWO(e) - \sum_{e \in \mathrm{L}} PWI(e)$ and $D_{\mathrm{L}} = \sum_{e \in \mathrm{L}} SFGW(e)$.

The flexibility of *Loop1* and *Loop2* can be determined as $F_1 = 2 \times 1 - 2 = 0$ and $F_2 = 2 \times 2 - (2 + 2 - 1 - 0) = 1$.

Step 1.3: Resolve Confict

The EMARS algorithm adopts the same principle as the MARS scheduling for resolving conflict (Wang and Parhi 1994). In the second-order IIR filter example, a valid conflict-free scheduling for recursive nodes is the same as the initial schedule shown in Table 9.7.

Step 2: Scheduling and Resource Allocation for Non-recursive Nodes

In this subsection, the main EMARS steps for scheduling and resource allocation of the non-recursive nodes are described.

Step 2.1: Calculate Minimum Number of Processors

The exact number of additional processors, $(Processor)_u$, can be calculated in MARS using the equation below.

$$(\text{Processor})_u = \left\lceil \frac{(N_u - TS_u) \times T_u}{P_u \times T} \right\rceil \tag{9.15}$$

where N_u is the number of type U operations within the non-recursive section, TS_u is the number of available time partitions in the U-type processors, T_u is the computation time of a U-type operation, P_u is the pipelining level of a U-type processor and T is the iteration period.

In EMARS, once again, T_u refers to the longest computation time of all outputs of a U-type operation and the P_u is the highest pipelining level of a U-type operation. For the second-order IIR filter, the exact number of additional MAC processors is given as:

$$(\text{Processor})_{\text{MAC}} = \left\lceil \frac{(3-4) \times 2}{2 \times 2} \right\rceil = 0 \quad (9.16)$$

There are four time partitions available for the MAC processor, and three time partitions needed for non-recursive MAC nodes. MARS determined that this filter will not require any new processors, therefore the number of MAC processors required will be three which is given in Equation (9.12).

Step 2.2: Locate Feed-forward Paths

The second-order IIR filter in Figure 9.13 only contains one feed-forward path shown in Figure 9.14, which ends on a recursive node. The recursive node MAC4 does not include in this feed-forward path.

Step 2.3: Create an Initial Schedule

Continuing with the second-order IIR filter shown in Figure 9.13, the initial scheduling from feed-forward paths is shown in Table 9.8, where M1–M3 refer to the three MAC processors. Note that from the schedule matrix, there exists no conflict and it is also the final *conflict-free* schedule for the non-recursive nodes.

Step 3: Control Circuit Implementation Using Revised Folding Technique

After a final schedule and the number of processors are given, IRIS can construct the datapaths that connect the processors, and generate the control circuitry using a revised folding transformation

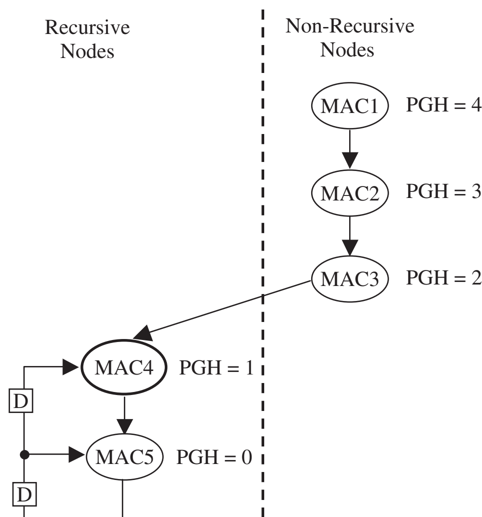


Figure 9.14 Schedule for a second order IIR filter

Table 9.8 Final *conflict-free* schedule for 2nd-order IIR filter

Time	MAC processors		
	M1	M2	M3
0	MAC5	$MAC3^{+1}$	$MAC1^{+2}$
1	$MAC4^{+1}$	$MAC2^{+1}$	

technique. The derivation of the revised folding equation, which forms the basis for the IRIS controller, is introduced in this subsection.

While the folding transformation reduces the number of functional units in the architecture, it may also lead to an architecture that uses a large number of registers. Some techniques can be used to minimize the number of registers (Parhi 1994), but they increase the routing circuit complexity. In this subsection, the hardware-sharing circuits are targeted to Xilinx FPGAs. As FPGAs are a 'register rich' architecture and the interconnect delay of the routed circuit can often be up to as much as 60% of the total critical path in an FPGA, there is a deliberate intention not to reduce registers as these can be used by retiming routine to improve throughout. These issues are highlighted in next section in the implementation of the delayed LMS filters (Yi and Woods 2006, Yi *et al*. 2005).

Step 3.1: Revised Folding Transformation

As illustrated in Section 8.6.2, a folding set is an ordered set of operations executed by the same functional unit (Parhi 1999). Each folding set contains N entries, some of which may be null operations. The operation in the jth position within the folding set (where j goes from 0 to $N-1$) is executed by the functional unit during the time partition j. The use of systematic folding techniques is demonstrated using the second-order IIR filter example. This filter is folded with folding factor $N = 2$, using folding sets that can be written as S1 = MAC5, MAC4, S2 = MAC3, MAC2 and S3 = MAC1, -. The folding sets S1, S2 and S3 contain only MAC operations, and the nodes in same folding set are executed by the same MAC hardware. The folding factor $N = 2$ means that the iteration period of the folding hardware is 2.

If a folded system can be realized, $D'_F(U \xrightarrow{e} V) \geq 0$ must hold for all of the edges in the SFG. Once valid folding sets have been assigned, retiming can be used to either satisfy this property or determine that the folding sets are not feasible. A set of constraints for each edge of the SFG is found using Equation (9.17) and the technique for solving systems of inequalities can be used to determine if a solution exists and to find a solution, if one indeed exists. The inequalities constraints is given below:

$$r(U) - rV \leq \lfloor \frac{D'_F(U \xrightarrow{e} V)}{N} \rfloor \tag{9.17}$$

where $\lfloor x \rfloor$ is the largest integer less than or equal to x.

In the following, the method by which these techniques can be used to design folded architectures and generate the folded second-order IIR filter is described. The basic procedure is as follows:

- *perform retiming for folding*
- *write the folding equations*
- *perform register allocation*
- *draw the folded architecture*

Table 9.9 Folding equations and retiming for folding constraints

Edge	Folding equation	Retiming for folding constraint
$1 \longrightarrow 2(P)$	$D'_{\mathrm{F}}(U \underset{\rightarrow}{e} V) = 2(1) - 0 + 0 + 1 - 0 = 3$	$r(1) - r(2) \leq 1$
$1 \longrightarrow 2(S)$	$D'_{\mathrm{F}}(U \underset{\rightarrow}{e} V) = 2(0) - 2 + 1 + 1 - 0 = 0$	$r(1) - r(2) \leq 0$
$2 \longrightarrow 3(P)$	$D'_{\mathrm{F}}(U \underset{\rightarrow}{e} V) = 2(1) - 0 + 0 + 0 - 1 = 1$	$r(2) - r(3) \leq 0$
$2 \longrightarrow 3(S)$	$D'_{\mathrm{F}}(U \underset{\rightarrow}{e} V) = 2(0) - 2 + 1 + 0 - 1 = -2$	$r(2) - r(3) \leq -1$
$3 \longrightarrow 4(S)$	$D'_{\mathrm{F}}(U \underset{\rightarrow}{e} V) = 2(0) - 2 + 1 + 1 - 0 = 0$	$r(3) - r(4) \leq 0$
$4 \longrightarrow 5(S)$	$D'_{\mathrm{F}}(U \underset{\rightarrow}{e} V) = 2(0) - 2 + 1 + 0 - 1 = -2$	$r(4) - r(5) \leq -1$
$5 \longrightarrow 5(S)$	$D'_{\mathrm{F}}(U \underset{\rightarrow}{e} V) = 2(1) - 2 + 0 + 0 - 0 = 0$	$r(5) - r(5) \leq 0$
$5 \longrightarrow 4(P)$	$D'_{\mathrm{F}}(U \underset{\rightarrow}{e} V) = 2(2) - 2 + 0 + 1 - 0 = 3$	$r(5) - r(4) \leq 1$

The folding equation (see Equation 8.10) and the retiming for folding constraints (see Equation 9.17) for the SFG in Figure 9.13(a) are given in the Table 9.9. According to the technique for solving systems of inequalities, the first step is to draw a constraint graph. Given a set of M inequalities in N variables where each inequality has the form $r_i - r_j \leq k$ for integer value of k, the constraint graph can be drawn using the following procedure (Parhi 1999):

- *draw the node i for each of the N variables* r_i;
- *for each inequalities* $r_i - r_j \leq k$, *draw the edge* $j \longrightarrow i$ *from the node* j *to the node* i *with length* k;
- *draw the node* $N + 1$, *for each node* $i, i = 1, 2, \ldots, N$, *draw the edge* $N + 1 \longrightarrow i$ *from the* $N + 1$ *to the node* i *with length 0.*

One of the shortest path algorithms (Parhi 1999) can be used to determine if a solution exists and to find a solution. The system of inequalities has a solution if and only if the constraint graph contains no negative cycles. If a solution exists, one solution is where r_i is the minimum-length path from the node $N + 1$ to the node i. The constraint graph for the inequalities in the right column of Table 9.9 is shown in Figure 9.15. The set of constraints has a solution because no negative cycles exist. One solution is $r(1) = -2, r(2) = -2, r(3) = -1, r(4) = -1, r(5) = 0$.

The retimed SFG is shown in Figure 9.16. The folding equation for the retimed SFG is given in Table 9.10, and the folded SFG is shown in Figure 9.17.

The revised folding transformations have been described in this section for mapping DSP algorithms to time-multiplexed architectures. The new folding equations suit both hierarchical and flattened circuits. For flattened circuits, Av will equal 0, and the folding equation will be the original one.

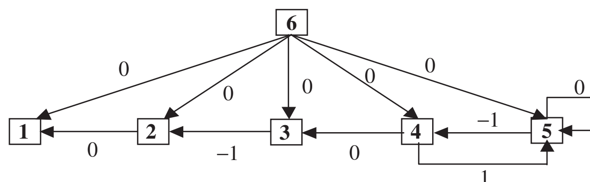


Figure 9.15 Constraints graph for the second-order IIR filter

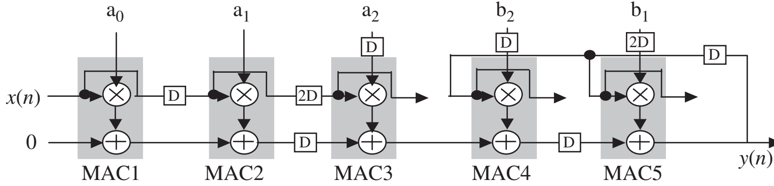


Figure 9.16 Retimed second-order IIR filter. Reproduced from *Hierarchical Synthesis of Complex DSP Functions Using IRIS* by Y. Yi & R. Woods, IEEE Trans on Computer Aided Design, Vol. 25, No. 5, © 2006 IEEE

Table 9.10 Folding equations for the retimed SFG in Figure 9.16

Edge	Folding equation
$1 \longrightarrow 2(P)$	$D'_F(U \xrightarrow{e} V) = 2(1) - 0 + 0 + 1 - 0 = 3$
$1 \longrightarrow 2(S)$	$D'_F(U \xrightarrow{e} V) = 2(0) - 2 + 1 + 1 - 0 = 0$
$2 \longrightarrow 3(P)$	$D'_F(U \xrightarrow{e} V) = 2(2) - 0 + 0 + 0 - 1 = 3$
$2 \longrightarrow 3(S)$	$D'_F(U \xrightarrow{e} V) = 2(1) - 2 + 1 + 0 - 1 = 0$
$3 \longrightarrow 4(S)$	$D'_F(U \xrightarrow{e} V) = 2(0) - 2 + 1 + 1 - 0 = 0$
$4 \longrightarrow 5(S)$	$D'_F(U \xrightarrow{e} V) = 2(1) - 2 + 1 + 0 - 1 = 0$
$5 \longrightarrow 5(S)$	$D'_F(U \xrightarrow{e} V) = 2(1) - 2 + 0 + 0 - 0 = 0$
$5 \longrightarrow 4(P)$	$D'_F(U \xrightarrow{e} V) = 2(1) - 2 + 0 + 1 - 0 = 1$
$a_0 \longrightarrow 1(Q)$	$D'_F(U \xrightarrow{e} V) = 2(0) - 0 + 0 + 0 - 0 = 0$
$a_1 \longrightarrow 2(Q)$	$D'_F(U \xrightarrow{e} V) = 2(0) - 0 + 0 + 1 - 0 = 1$
$a_2 \longrightarrow 3(Q)$	$D'_F(U \xrightarrow{e} V) = 2(1) - 0 + 0 + 0 - 0 = 2$
$b_2 \longrightarrow 4(Q)$	$D'_F(U \xrightarrow{e} V) = 2(1) - 0 + 0 + 1 - 0 = 3$
$b_1 \longrightarrow 5(Q)$	$D'_F(U \xrightarrow{e} V) = 2(2) - 0 + 0 + 0 - 0 = 4$

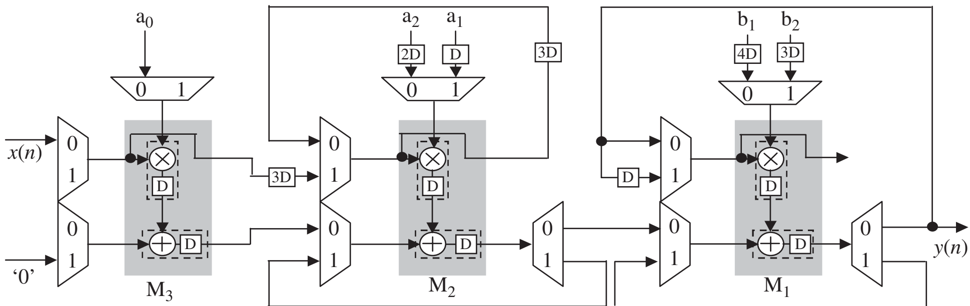


Figure 9.17 Folded second-order IIR filter. Reproduced from *Hierarchical Synthesis of Complex DSP Functions Using IRIS* by Y. Yi & R. Woods, IEEE Trans on Computer Aided Design, Vol. 25, No. 5, © 2006 IEEE

9.6 Case Study: Adaptive Delayed Least-mean-squares Realization

Adaptive filters have uses in a number of applications that require differing filter characteristics in response to variable signal conditions, such as noise cancellation, linear prediction, adaptive signal enhancement, and adaptive control. The LMS algorithm is the most widely used adaptive filtering algorithm in practice (Farhang-Boroujeny 1998). The wide spectrum of applications of the LMS algorithm can be attributed to its simplicity and robustness to signal statistic. The TF-DLMS algorithm is described by the following equations:

Filter output:

$$y_n = \sum_{i=0}^{N-1} w_{n-i,i} x_{n-i} = w_{n,0} x_n + w_{n-1,1} x_{n-1} + \cdots + w_{n-(N-1),N-1} x_{n-(N-1)} \tag{9.18}$$

Error:

$$e(n-m) = d(n-m) - y(n-m) \tag{9.19}$$

Delayed coefficient update:

$$w_{n,i} = w_{n-1,i} + 2\mu e_{n-m} x_{n-m-i} \tag{9.20}$$

An 8-tap TF-DLMS architecture is given in Figure 9.18. Using the DLMS algorithm, registers can be inserted into the error feedback path before the adaptation loop. The main challenge now is to use these delays as a means of pipelining the LMS filter. This process first involves the determination of the number of delays needed to achieve a fully pipelined version of the circuit. This is important as a value that is too small will lead to a low-speed circuit, whilst too large a value will produce a slower convergence rate and poor tracking capacity. IRIS determines the unknown amount of delays (e.g. mD) in terms of circuit performance (speed and area) requirements. Once this has been determined, a fully pipelined implementation is developed.

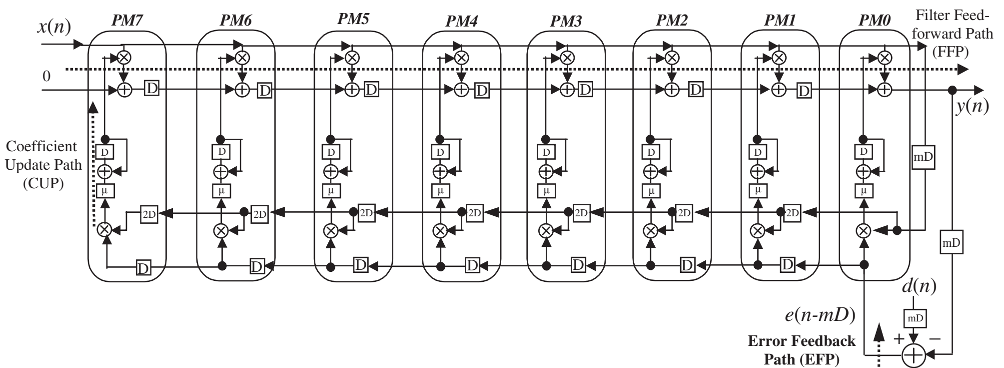


Figure 9.18 An 8-tap TF-DLMS algorithm architecture

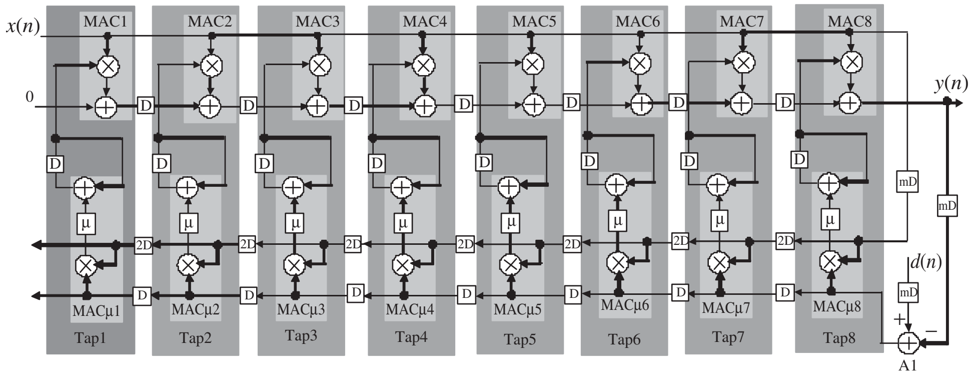


Figure 9.19 The 8-tap TF-DLMS filter with m delays in EFP datapath. Reproduced from *Hierarchical Synthesis of Complex DSP Functions Using IRIS* by Y. Yi & R. Woods, IEEE Trans on Computer Aided Design, Vol. 25, No. 5, © 2006 IEEE

9.6.1 High-speed Implementation

The TF-LMS adaptive filter (Jones 1992) has similar convergence behaviour as the DLMS filter. An 8-tap predictor system with mD delays is given in Figure 9.18. The number of delays has been defined as mD as this value will be determined by the number of pipeline stages (mD) which has not yet been determined. This emphasizes how choices at the circuit architectural level impacts the design of the algorithm.

The hierarchical SFG of the 8-tap TF-DLMS with m delays in the EFP datapath is given in Figure 9.19. It shows that the DLMS filter is constructed from an adder and eight instances of tap cell. Within these tap cells, one MAC and several MACμ (a MAC with a scaling circuit μ) operations are carried out to implement the multiplier and accumulate function.

The arithmetic representation of a circuit is a three-level hierarchical circuit. The types of processor chosen for this design example include pipelined dedicated multiplier and adder processors. The DLMS filter was implemented using fixed-point arithmetic and all fixed-point implementation issues were considered (Ting *et al*. 2000). This analysis includes: a detailed study of wordlength growth in order to achieve the best wordlength in terms of the adaptive filtering performance and cost of realization, application of truncation/rounding circuitry, use of saturation, and exploitation of bit-shift for simple multiplication (Ting *et al*. 2000).

If the given specification demands a high-speed circuit which we assume here as an FPGA implementation is being targeted, the multipliers and adders will be pipelined. In this design, the multiplier operation in the MAC and $MAC\mu$ subsystems is based on a dedicated 8-bit signed multiplier, which is pipelined by one stage. The circuit for the adder processor is based on an addition of two 8-bit and 16-bit words in the MAC and MACμ subsystem respectively, and can be implemented using a fast carry chain adder pipelined by one stage. The scaling circuit is implemented using a 5-bit arithmetic shift right operation. The step size used in the RDLMS design is 2^{-5} (0.03125) which is the nearest power-of-2 number to the optimal step size obtained using Matlab^{TM} simulation.

Changing the latency of the multiplier and adder to one delay gives the circuit architecture in Figure 9.20. After the IRIS SignalCompiler reads the model file of the 8-tap TF-DLMS adaptive filter, graphical representations of the basic parameterized processor models (multiplier and adder), displaying the information contained within the IRIS processor library, are generated. They are also shown in Figure 9.20. All input and output data time formats and the output latency of basic

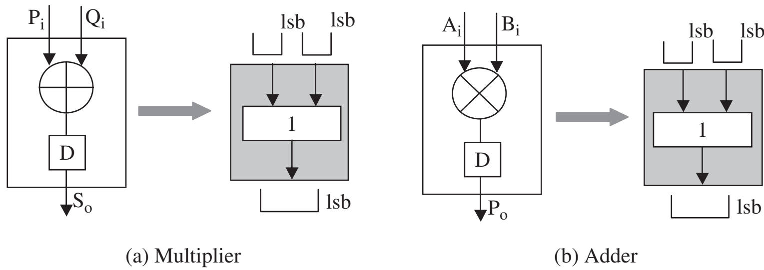


Figure 9.20 Multiplier and adder models. Reproduced from *Hierarchical Synthesis of Complex DSP Functions Using IRIS* by Y. Yi & R. Woods, IEEE Trans on Computer Aided Design, Vol. 25, No. 5, © 2006 IEEE

Table 9.11 Data time format and latency of multiplier (S=A*B)

Port name	Data time format		Latency
	Type	Value	
A	Parallel	[0,0,0,0,0,0,0,0]	0
B	Parallel	[0,0,0,0,0,0,0,0]	0
S	Parallel	[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]	1

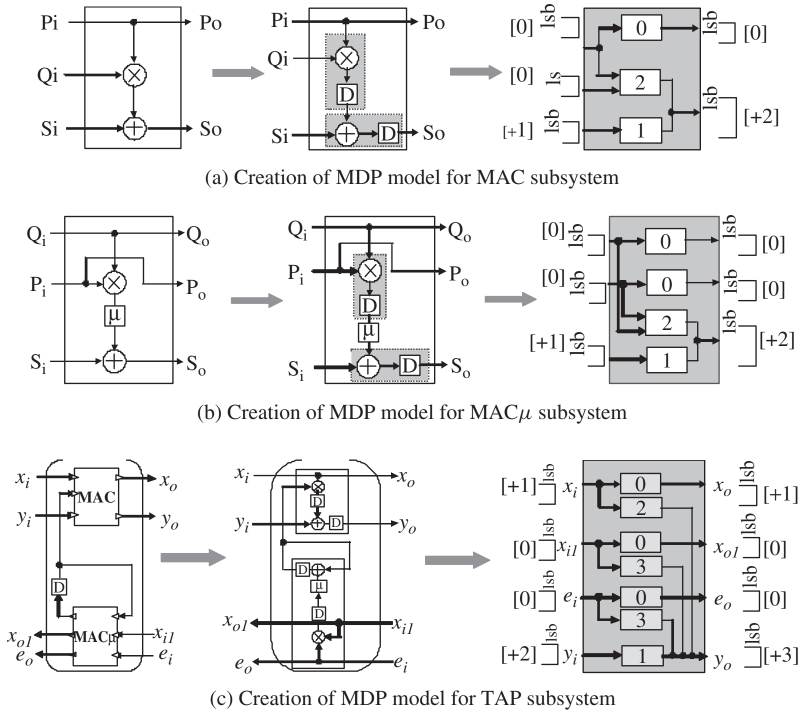
Table 9.12 Data time format and latency of adder (S=A+B)

Port name	Data Time Format		Latency
	Type	Value	
A	Parallel	[0,0,0,0,0,0,0,0](8-bit) [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] (16-bit)	0
B	Parallel	[0,0,0,0,0,0,0,0](8-bit) [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] (16-bit)	0
S	Parallel	[1,1,1,1,1,1,1,1](8-bit) [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1] (16-bit)	1

Xilinx processor blocks used in this design are given in Tables 9.11 and 9.12. Parameter values for the nodes shown in Figure 9.21 need to be supplied to calculate specific latency values and data time shapes for the filter design. In this example, the automatic synthesis process for hierarchical design is implemented using the IRIS tools described in Section 9.4.

Since there is no feedback loop in the subsystems of MAC and MACμ, the pipelining period is 1. IRIS automatically synthesizes the MAC and MACμ subsystems using the retiming technique described in Section 9.4, and the truncation question is also considered using this method. The original circuit, synthesized circuit and the MDP models for the MAC and MACμ subsystems are shown in Figure 9.21(a) and (b), respectively. All inputs and outputs data time formats and output latency are given in Tables 9.13 and 9.14. The relationship inputs vectors are shown in Table 9.15.

The generated MDP models of subsystems MAC and MACμ are used in the higher-level subsystem synthesis. The higher-level subsystem is a TAP subsystem and its pipelining period is 1.



(a) Creation of MDP model for MAC subsystem

(b) Creation of MDP model for MACμ subsystem

(c) Creation of MDP model for TAP subsystem

Figure 9.21 Subsystem models in TF-DLMS filter. Reproduced from *Hierarchical Synthesis of Complex DSP Functions Using IRIS* by Y. Yi & R. Woods, IEEE Trans on Computer Aided Design, Vol. 25, No. 5, © 2006 IEEE

Table 9.13 The data time format and latency of the MAC Subsystem

Port name	Data time format		Latency
	Type	Value	
P_i	Parallel	[0,0,0,0,0,0,0,0]	
Q_i	Parallel	[0,0,0,0,0,0,0,0]	
S_i	Parallel	[1,1,1,1,1,1,1,1]	
P_o	Parallel	[0,0,0,0,0,0,0,0]	0
S_o	Parallel	[2,2,2,2,2,2,2,2]	2

The original circuit, synthesized circuit and the MDP model of the TAP subsystem are shown in Figure 9.21(c). All inputs and outputs data time formats and the output latency of TAP are given in Table 9.16. The relationship inputs vectors of subsystem TAP have been shown in Table 9.15. The datapath delay pairs of TAP subsystem, which include the function delay, are shown in Table 9.17.

The top-level 8-tap TF-DLMS circuit is synthesized using the MDP models of the TAP subsystem and adder processor. The number of pipeline cuts in the error feedback (mD) of TF-DLMS

Table 9.14 The data time format and latency of MACμ Subsystem

Port name	Data time format		Latency
	Type	Value	
P_i	Parallel	[0,0,0,0,0,0,0,0]	
Q_i	Parallel	[0,0,0,0,0,0,0,0]	
S_i	Parallel	[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]	
P_o	Parallel	[0,0,0,0,0,0,0,0]	0
Q_o	Parallel	[0,0,0,0,0,0,0,0]	0
S_o	Parallel	[2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2]	2

Table 9.15 Relationship inputs vectors of MAC, MAC and TAP

Module output	MAC	MACμ	Module output	TAP
P_o	$[P_i]$	$[P_i]$	x_0	$[x_i]$
Q_o		$[Q_i]$	x_{0l}	$[x_{il}]$
S_o	$[P_i, Q_i, S_i]$	$[P_i, Q_i, S_i]$	e_0	$[e_i]$
			y_0	$[x_i, x_{il}, e_i, y_i]$

Table 9.16 The data time format and latency of TAP

Port name	Data time format		Latency
	Type	Value	
x_i	Parallel	[1,1,1,1,1,1,1,1]	
x_{il}	Parallel	[0,0,0,0,0,0,0,0]	
e_i	Parallel	[0,0,0,0,0,0,0,0]	
y_i	Parallel	[2,2,2,2,2,2,2,2]	
x_o	Parallel	[1,1,1,1,1,1,1,1]	1
x_{ol}	Parallel	[0,0,0,0,0,0,0,0]	0
e_o	Parallel	[0,0,0,0,0,0,0,0]	0
y_o	Parallel	[3,3,3,3,3,3,3,3]	3

Table 9.17 Datapath delay pair of TAP subsystem

	TAP			
	x_i	x_{il}	e_i	y_i
x_o	(0,0)			
x_{ol}		(0,0)		
e_o			(0,0)	
y_o	(2,0)	(4,1)	(4,1)	(1,0)

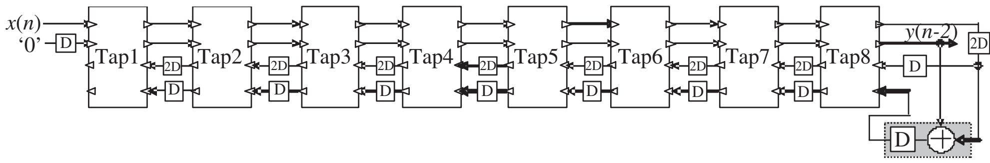


Figure 9.22 The synthesized TF-RDLMS filter. Reproduced from *Hierarchical Synthesis of Complex DSP Functions Using IRIS* by Y. Yi & R. Woods, IEEE Trans on Computer Aided Design, Vol. 25, No. 5, © 2006 IEEE

circuit needs to be determined. Eight loops are identified in the top-level 8-tap TF-DLMS architecture as shown below. The corresponding pipelining period is calculated using Equation (9.16). Pipelining periods of 1–8 are determined and made equal to 1 cycle in order to generate a high-speed architecture. The optimal values for these loops are given as: $m_8 = -3D, m_7 = -2D, m_6 = -1D, m_5 = 0D, m_4 = 1D, m_3 = 2D, m_2 = 3D, m_1 = 4D$. From this equation, it can be seen that the optimal value of the number of delays m for the complete architecture is determined by the slowest cycle and equals $4D$. The synthesized circuit for the 8-tap predictor filter using IRIS is shown in Figure 9.22, where the synthesized tap architecture in shown in Figure 9.21(c).

Loop 1: $\text{TAP8}(y_o) \longrightarrow mD \longrightarrow A1 \longrightarrow \text{TAP8}(e_i)$

$$\alpha_1 = \left\lceil \frac{4+1-0-0}{m+1} \right\rceil = 1 \Longrightarrow m_1 = 4D$$

Loop 2: $\text{TAP7}(y_o) \longrightarrow \text{TAP8}(y_o) \longrightarrow mD \longrightarrow A1 \longrightarrow \text{TAP8}(e_o) \longrightarrow \text{TAP7}(e_i)$

$$\alpha_2 = \left\lceil \frac{4+4+1+0-2-0-0-0}{1+1+m+2} \right\rceil = 1 \Longrightarrow m_2 = 3D$$

Loop 3: $\text{TAP6}(y_o) \longrightarrow \text{TAP7}(y_o) \longrightarrow \text{TAP8}(y_o) \longrightarrow mD \longrightarrow A1 \longrightarrow \text{TAP8}(e_o) \longrightarrow TAP7(e_o) \longrightarrow TAP6(e_i)$

$$\alpha_3 = \left\lceil \frac{4+4+4+1+0+0-2-2-0-0-0-0}{1+1+1+m+4} \right\rceil = 1 \Longrightarrow m_3 = 2D$$

⋮

Loop 8: $\text{TAP1}(y_o) \longrightarrow \text{TAP2}(y_o) \longrightarrow \text{TAP3}(y_o) \longrightarrow \text{TAP4}(y_o) \longrightarrow \text{TAP5}(y_o) \longrightarrow \text{TAP6}(y_o) \longrightarrow \text{TAP7}(y_o) \longrightarrow \text{TAP8}(y_o) \longrightarrow mD \longrightarrow A1 \longrightarrow \text{TAP8}(e_o) \longrightarrow \text{TAP7}(e_o) \longrightarrow \text{TAP6}(e_o) \longrightarrow \text{TAP5}(e_o) \longrightarrow \text{TAP4}(e_o) \longrightarrow \text{TAP3}(e_o) \longrightarrow \text{TAP2}(e_o) \longrightarrow \text{TAP1}(e_i)$

$$\alpha_8 = \left\lceil \frac{4\times 8+1+0\times 7-2\times 7-0-0\times 6}{1\times 8+m+14} \right\rceil = 1 \Longrightarrow m_8 = -3D$$

$$m = \max(allm_c) = 4D$$

The TF-RDLMS N-tap filter has been constructed using the TAP processor module structure shown in Figure 9.23. In this TAP structure, the filter weights are updated locally, therefore the order can be increased by adding more TAP elements, if needed. The filter also has the advantage that increasing the filter order does not increase the critical path. The latency of the error signal $e(n)$, which is used to update the filter coefficient and affects filter performance, is increased by 4 delays rather than by $3N$ delays as in the TF-FPDLMS structure (Ting *et al*. 2000) which provides a performance advantage.

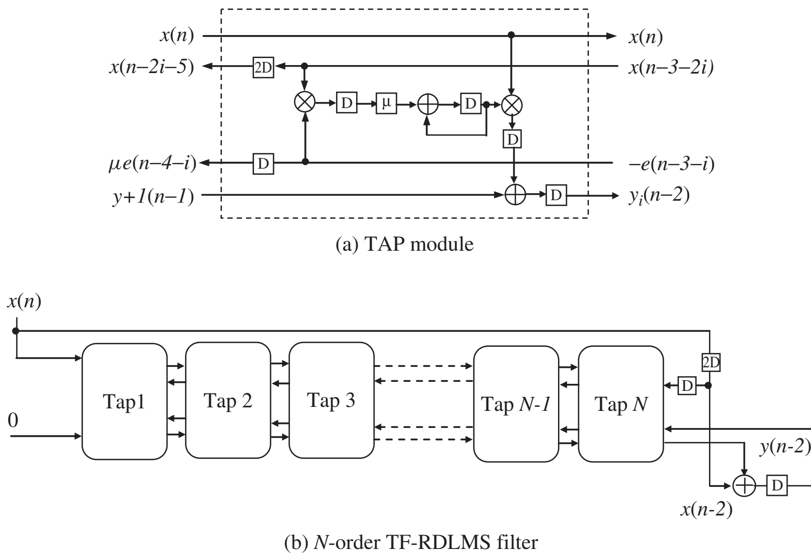


Figure 9.23 Transposed fine-grain retiming DLMS filter

9.6.2 Hardware-shared Designs for Specific Performance

Due to the advance of the Xilinx FPGA Virtex-II technology, the speed of the dedicated adder and multiplier has up to 180 MSPS performance. In many applications, sampling rates can exceed expectations, so hardware sharing may be desirable. To achieve this, the EMARS scheduling algorithm and revised folding technique is used. Here, the hardware sharing circuit for an 8-tap TF-DLMS filter is generated using the IRIS scheduling functions. The ratio between the clock rate and the sampling rate can be regarded as a hardware sharing factor (HSF):

$$HSF = \frac{clock\ \ rate}{sampling\ \ rate}$$

which is also equal to the pipelining period. Assuming that the circuit sampling rate is 80 MHz, the hardware sharing factor is then 2, which is $\lfloor 180/80 \rfloor = 2$.

As the subsystems of MAC and MAC do not include the feedback loop and delays, the synthesized circuit and MDP model with pipelining period of 2 will be the same as the circuit with the pipelining period of 1 shown in Figure 9.21(a) and (b). The synthesized circuit of the TAP subsystem, and the corresponding IRIS MDP Model with pipelining period of 2 are shown in Figure 9.24. All inputs and outputs data time formats and the TAP output latency with pipelining period of 2 are given in Table 9.18. The relationship input vectors of the TAP subsystem are the same as for the TF-DLMS filter shown in Table 9.15. The datapath delay pairs of the TAP subsystem with sharing circuit is shown in Table 9.19.

The top-level 8-tap TF-DLMS circuit is synthesized using this TAP subsystem MDP model and adder processors. The number of pipeline-cuts in the error feedback (mD) needs to be decided. The same method is used to decide the optimal m values for the TF-DLMS circuit in Figure 9.19. In this case, the optimal value of m is 2 because the pipelining period is 2. The final conflict-free scheduling matrix for the TF-DLMS is shown in Table 9.20. Note that iteration period is equal

to the pipelining period in this matrix. However, a new processor was allocated because of the tight precedence constraints. The final hardware operators counted for this example equal the exact number of four TAP operators (T1–T4) and one addition operator (A1).

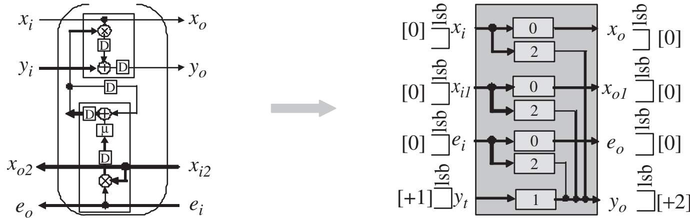


Figure 9.24 The synthesized tap subsystem and MDP models ($\alpha = 2$)

Table 9.18 The data time format and latency of TAP

Port name	Data time format		Latency
	Type	Value	
x_i	Parallel	[0,0,0,0,0,0,0,0]	
x_{il}	Parallel	[0,0,0,0,0,0,0,0]	
e_i	Parallel	[0,0,0,0,0,0,0,0]	
y_i	Parallel	[1,1,1,1,1,1,1,1]	
x_o	Parallel	[0,0,0,0,0,0,0,0]	0
x_{ol}	Parallel	[0,0,0,0,0,0,0,0]	0
e_o	Parallel	[0,0,0,0,0,0,0,0]	0
y_o	Parallel	[2,2,2,2,2,2,2,2]	2

Table 9.19 Datapath delay pair of TAP subsystem

	TAP			
	x_i	x_{il}	e_i	y_i
x_o	(0,0)			
x_{ol}		(0,0)		
e_o			(0,0)	
y_o	(2,0)	(4,2)	(4,12)	(1,0)

Table 9.20 The scheduling matrix of a hardware-sharing TF-DLMS filter

Time step	TAP				Adder
	T1	T2	T3	T4	A1
0	Tap1	Tap2	Tap3	Tap4	Adder
1	Tap5	Tap6	Tap7	Tap8	

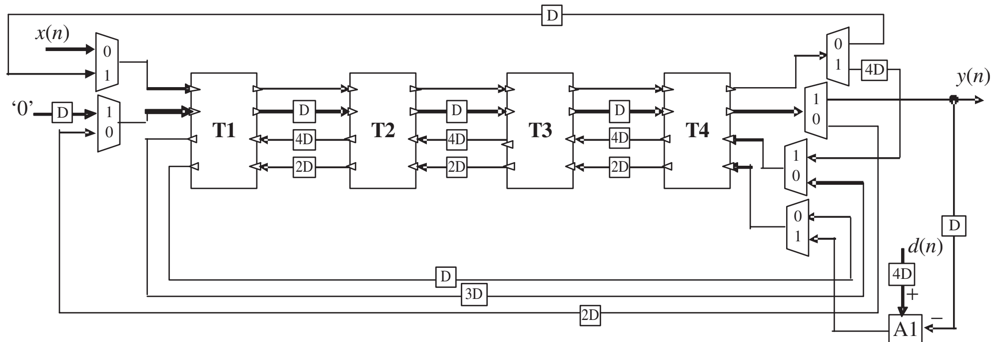


Figure 9.25 The final hardware architecture of the TF-DLMS filter of Figure 9.5 as generated by IRIS with pipelining period of 2

The hardware sharing circuit for the TF-DLMS circuit with pipelining period of 2, using the IRIS scheduling function is shown in Figure 9.25.

9.7 Conclusions

The material presented here describes work undertaken in creating a SFG-based synthesis tool, IRIS which has been used to automate many of the techniques presented in Chapter 8. In addition, the material has shown how the hierarchical nature of design creation, places considerable design constraints on the techniques presented there and additional criteria need to be taken into consideration to create the hardware cores. In the chapter, two examples, namely the second-order IIR filter and the fifth-order WDE filter have been used to demonstrate the challenges. This provides a clear insight into the challenges that will be met when creating FPGA functionality. Chapter 12 now gives details on the additional aspects required to capture this functionality in the form of a useful IP core.

References

AccelFPGA (2002) Accelfpga: High-level synthesis for dsp design. Web publication downloadable from http://www.accelchip.com/.

Bellows P and Hutchings B (1998) Jhdl – an hdl for reconfigurable systems. *IEEE Symposium on FPGA's for Custom Computing Machines*, pp. 175–184, Napa, USA.

Bringmann O and Rosenstiel W (1997) Resource sharing in hierarchical synthesis. *Int. Conf. on Computer Aided Design*, pp. 318–325.

C. E. Leiserson FMR and Saxe JB (1983) Optimizing synchronous circuitry by retiming. *Proc. 3rd Caltech Conference on VLSI*, pp. 87–116.

Christofides N (1975) *Graph Theory – An Algorithmic Approach*, Academic Press, London.

Davidson S, Landskov D, Shriver B and Mallett PW (1981) Some experiments in local microcode compaction for horizontal machines. *IEEE Trans. Computers*, pp. 460–477.

Derrien S and Risset T (2000) Interfacing compiled fpga programs: the mmalpha approach. *Journal of Parallel and Distributed Processing Techniques and Applications*.

Dewilde P, Deprettere E and Nouta R (1985) *Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms*, Prentice Hall, pp. 258–264.

Drost A 1999 Hierarchy management in synplify. Web publication downloadable from http://www.xilinx.com/.

Farhang-Boroujeny B (1998) *Adaptive Filters Theory and Applications*. John Wiley & Sons Inc.

Gazsi L (1985) Explicit formulas for lattice wave digital filters. *IEEE Trans. Circuits and Systems* **CAS-32**, 68–88.

Gnizio JP (1985) *Introduction to Linear Goal Programming*. Sage Publications, Beverly Hills.

Jones DL (1992) Learning characteristics of transpose-form lms adaptive filters. *IEEE Trans. Circuits and Systems-II: Analog and Digital Signal Proc.* **39**, 745–749.

Keating M and Bricaud P (1998) *Reuse Methodology Manual for System-On-A-Chip Designs*. Kluwer Academic Publishers, Norwell, MA, USA.

Kung SY (1988) *VLSI Array Processors*. Prentice Hall, Englewood Cliffs, NJ.

Lawson S and Mirzai A (1990) *Wave Digital Filters*. Ellis Horwood, New York.

Lee J, Hsu Y and Lin Y (1989) A new integer linear programming formulation for the scheduling problem in data-path synthesis *Proc. Int. Conf. on Computer Aided Design*, pp. 20–23.

McCanny JV, Ridge D, Hu Y and Hunter J (1997) Hierarchical vhdl libraries for dsp asic design *Proc Int. Conf. on Acoustics, Speech and Signal Processing*, Munich, pp. 675–678.

McGovern B (1993) *The Systematic Design of VLSI Signal Processing Architectures*. PhD dissertation School of Electrical and Electronic Engineering, Queen's University of Belfast.

Parhi KK (1994) Calculation of minimum number of registers in arbitrary life time chart. *IEEE Trans. Circuits and Systems-II* **41**(6), 434–436.

Parhi KK (1999) *VLSI digital signal processing systems : design and implementation*. John Wiley & Sons, Inc., New York.

Park IC and Kyung CM (1991) Fast and near optimal scheduling in automatic data path synthesis *Proc. 28th DAC*, pp. 680–685.

Paulin PG and Knight JP (1989) Force directed scheduling for the behavioural synthesis of asic's. *IEEE Trans. Computer Aided Design* **8**, 661–679.

Roy J (1993) *Parallel Algorithms For High-Level Synthesis*. PhD dissertation the University of Cincinnati.

Synplify (2003) Synplify pro: the industry #1 fpga synthesis solution. Web publication downloadable from http://www.synplicity.com/.

Ting L, Woods R, Cowan C, Cork P and Sprigings C (2000) High-performance fine-grained pipelined lms algorithm in virtex fpga *Advanced Signal Processing Algorithms, Architectures, and Implementations X: SPIE San Diego*, pp. 288–299.

Trainor DW (1995) *An Architectural Synthesis Tool for VLSI Signal Processing Chips.* PhD dissertation School of Electrical and Electronic Engineering, Queen's University of Belfast.

Trainor DW, Woods RF and McCanny JV (1997) Architectural synthesis of digital signal processing algorithms using iris. *Journal of VLSI Signal Processing* **16**(1), 41–56.

Vajda S (1981) *Linear Programming: Algorithms and Applications*. Chapman and Hall, London.

Vanhoof J, Rompaey KV, Bolsens I, Goossens G and De Man H (1993) *High-Level Synthesis for Real-Time Digital Signal Processing*. Kluwer Academic Publishers, Dordrecht/Boston/London.

Wang CY and Parhi KK (1994) The MARS High-Level DSP Synthesis System. *VLSI Design Methodologies for Digital Signal Processing Architectures*, Kluwer.

Xilinx Inc. (1999) Using xilinx and synplify for incremental designing (ceo). Web publication downloadable from http://www.xilinx.com/xapp/xapp164.pdf. Xilinx Application Notes.

Xilinx Inc. (2000) Xilinx system generator v2.1 for simulink reference guide. Web publication downloadable from http://www.xilinx.com.

Xilinx Inc. (2001) Xilinx software manual: Design manage / flow engineer guide. Web publication downloadable from http://toolbox.xilinx.com/docsan.

Yi Y and Woods R (2006) Hierarchical synthesis of complex dsp functions using iris. *IEEE Trans. Computer Aided Design*, **25**(5), 806–820.

Yi Y, Woods R, Ting L and Cowan C (2002) Implementing high-speed delayed-lms filter using the virtex-ii fpg *Proc. IEE Non-Linear and Non-Gaussian Signal Processing - N2SP Workshop*, Peebles, London.

Yi Y, Woods R, Ting L and Cowan C 2005 High speed fpga-based implementations of delayed-lms filters. *J. VLSI Signal Processing* **39**(1-2), 113–131.

10

Complex DSP Core Design for FPGA

Silicon technology is now at the stage where it is feasible to incorporate many millions of gates on a single square centimetre of silicon (Rowen 2002) with predictions of chip designs reaching over 2 billion transistors/cm^2 by 2015 (Soderquist and Leeser 2004). This now permits extremely complex functions, which would previously be implemented as a collection of individual chips, to be built as a complete system-on-a-chip (SoC). The ability to encompass all parts of an application on the same piece of silicon presents advantages of lower power, greater reliability and reduced cost of manufacture. Consequently, increased pressure has been put on designers to meet ever-tightening time-to-market deadlines, now measured in months rather than years. The whole emphasis within the consumer market is to have high numbers of sales of cheaper products with a slimmer life span and in a much shorter time-to-market. Particularly with the onset of new standards such as H.264 and MPEG4 which are driving the growth of high-definition TV and mobile video.

This increasing chip density has enabled an expansion of FPGA capabilities with devices such as Xilinx's Virtex V and Altera's Stratix III offering full SoC functionality. With their vast expanse of usable gates comes the problem of developing increased complex systems to be implemented on these devices. Just as with ASIC development, as the complexity of the designs rises, the difference in the growth rate of the feasible number of gates per chip and the number of gates per chip actually being manufactured, widens. As was indicated in Chapter 1, this is commonly referred to as the design productivity gap (IRTS 1999) and highlights a divergence that will not be closed by incremental improvements in design productivity. Instead a complete shift in the methodology of designing and implementing multi-million gate chips is needed that will allow designers to concentrate on higher levels of abstraction within the designs.

As the silicon density grows, design complexity increases at a far greater rate since they are now composed of more facets of the full system design and may combine components from a range of technological disciplines. Working more at the system level, designers are now becoming more heavily involved with integrating the key components without the freedom to delve deep into the design functionality. Existing design and verification methodologies have not progressed at the same pace, consequently adding to the widening gap between design productivity and silicon fabrication capacity.

Test and verification has become a major aspect of electronic design under much concern at present. Verification of such complex systems has now become the bottleneck in system-level

FPGA-based Implementation of Signal Processing Systems R. Woods, J. McAllister, G. Lightbody and Y. Yi
© 2008 John Wiley & Sons, Ltd

design as the difficulties scale exponentially with chip complexity. Design teams may often spend as much as 90% of their development effort on block or system level verification (Rowen 2002). There are many strategies being investigated to develop systems to accelerate chip testing and verification, particularly to address the increased difficulty in testing design components integrated from a third party. So much more is at stake, both with time and monetary concerns. The industry consensus on the subject is well encapsulated by Rowen (2002):

"Analysts widely view earlier and faster hardware and software validation as a critical risk-reducer for new product development projects".

This chapter will cover the evolution of reusable design processes, with a focus on FPGA-based IP core generation. Issues such as the development of reusable IP cores through to integration issues are discussed. The chapter starts in Section 10.1 by outlining the motivation for *design for reuse* and giving some targets for reuse. Section 10.2 outlines the development of IP cores and Section 10.3 highlights how they have grown from simple arithmetic libraries to complex system components. This is followed by a description of how parameterizable IP cores are created in Section 10.4 with the FIR filter used to demonstrate some of the key stages. The integration process is covered in Section 10.5 and followed by a description of a case study, namely the ADPCM IP core in Section 10.6. Finally, a brief description is given of the third party FPGA IP vendors in Section 10.7 followed by some conclusions in Section 10.8.

10.1 Motivation for Design for Reuse

There is a great need to develop design and verification methodologies that will accelerate the current design process so that the design productivity gap can be narrowed. As the number of available transistors doubles with every technology cycle then the target should be to increase design productivity at this same rate. To enable such an achievement, a greater effort is needed to research the mechanics of the design, test, and verification processes, an area that to date has so often been neglected. *Design for reuse* is heralded to be one of the key drivers in enhancing productivity, particularly aiding system-level design.

In addition to exponentially increased transistor counts, the systems themselves have become increasingly complex, due to the combination of complete systems on a single device with component heterogeneity bringing with it, a host of issues regarding chip design, and in particular test and verification. Involving full system design means that developers need to know how to combine all the different components building up to a full system-level design. The sheer complexity of this development process impacts the design productivity and creates ever-demanding time-to-market deadlines. Obtaining a balance of design issues such as performance, power management and manufacturability, with time-to-market and productivity is a multidimensional problem that is increasing in difficulty with each technological advancement.

Enhancement in design productivity can be achieved by employing design for reuse strategies throughout the entire span of the project development from initial design through to functional test and final verification. By increasing the level of abstraction, the design team can focus on pulling together the key components of the system-level design, using a hierarchical design approach. This same approach needs to be employed through all the modelling, simulation, test and verification of component and higher-level blocks. For verification there needs to be a formal strategy with code reuse and functional test coverage (Huang *et al*. 2001, Moretti 2001, Varma and Bhatia 1998).

To increase the overall productivity and keep pace with each technology generation, the amount of reuse within a system design must increase at the same rate, and the level of abstraction must rise. Productivity gains by employing reuse strategies for high-level functional blocks are estimated

to be in excess of 200% (Semiconductor Industry Association 2005). These reusable components need to be pre-verified with their own independent test harness that can be incorporated into the higher-level test environment. This can be achieved by incorporating IP cores from legacy designs or third party vendors. The need for such cores has driven the growth in IP core market, with ever greater percentages of chip components coming from IP cores.

Within the International Technology Roadmap for Semiconductors 2005 report (Semiconductor Industry Association 2005) the percentage of logic from reused blocks is currently at 36% (2008) and this figure is expected to steadily increase to 48% by 2015, and continue increasing in a similar manner beyond this time frame. The ITRS has published an update to the 2005 roadmap giving market-driven drivers (Association SI 2006). Table 10.1 gives a summary of some of the data referring to the need for reuse.

Another important area for improving design productivity is by forming the same design for reuse principles with embedded software, resulting in libraries of functions that can be integrated into the system-level design.

10.2 Intellectual Property (IP) Cores

One of the most favourable solutions for enhancing productivity is the strategy of using pre-designed functional blocks known as silicon intellectual property (IP) cores, often referred to as virtual circuits (VCs). The terminology of IP cores applies to a range of implementations, from dedicated circuit layout designs through to efficient code targeted to programmable DSP or RISC processors, or core descriptions captured in a hardware description language (HDL). Within the realms of ASIC and FPGA implementations, IP cores are often partitioned into three categories:

- hard IP
- firm IP
- soft IP

Hard IP refers to designs represented as mask layouts, whereas firm IP refers to synthesized netlists for a particular technology. Soft IP refers to the HDL version of the core that will have scalability and parameterization built in. For the latter, the term that has evolved is *parameterizable IP*. They can be designed so that they may be synthesized in hardware for a range of specifications and processes. For digital signal processing (DSP) applications parameters such as filter tap size, transform point size, or wordlength (Erdogan *et al*. 2003, Guo *et al*. 2004, Hunter 1999, McCanny *et al*. 1996), may be made to be a programmable feature. Parameters controlling these features would be fed into the code during the synthesis, resulting in the desired hardware for the application. There are advantages and disadvantages with each type of IP core, as illustrated in Figure 10.1.

Table 10.1 SOC design productivity trends (normalized to 2005)

	2005	’08	’10	’12	’14	’16	’18	’20
Design needed to be reused (%)	30	42	50	58	66	74	82	90
Trend: SOC total logic size	1.0	2.2	3.4	5.5	8.5	13.8	20.6	34.2
Required productivity for new designs	1.0	2.0	3.0	4.6	6.7	10.2	14.3	22.1
Required productivity for reused designs	2.0	4.0	6.0	9.2	13.5	20.4	28.6	44.2

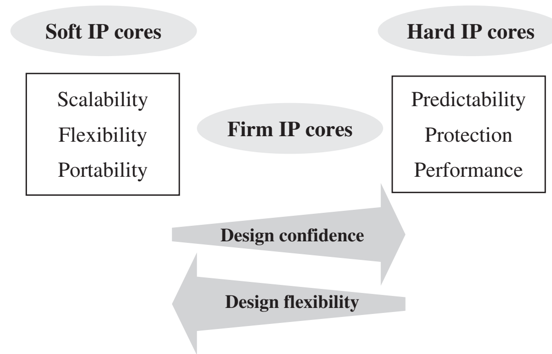


Figure 10.1 Benefits of IP types

These types of parameterizable soft IP cores allow the most flexibility and will provide the best levels of reuse, but there is a major cost in terms of test and verification as they will need to be tested in every mode. Moreover, there is an presumption that the cores will work best across different FPGA technologies or that the performance will scale linearly as the parameters are changed within the current FPGA technology. These issues thus act to reduce design confidence. These problems do not arise with hard IP cores such as the PowerPC in the Xilinx FPGAs, where the dedicated functionality is predictable. However, the major limitation here is that this platform is not suitable for DSP applications, as the concurrency offered by the FPGA was the main reason for opting for the FPGA platform in the first place. Thus, the flexibility is extremely limited by the underlying dedicated hardware platform.

Although fixed in their design, some flexibility can still be included from the onset in firm and hard IP devices. In these cases, the IP parameters that define the core are termed static IP (Junchao *et al*. 2001), whereby registers internal to the final design can be set to allow a multiplexing of internal circuits so to reconfigure the functionality of the design. Reconfiguration has been a subject of great interest with FPGAs, particularly with their increasing capabilities, (Alaraje and DeGroat 2005, Sekanina 2003). In contrast, the IP parameters within soft IP cores are termed dynamic IP parameters. They are often local or global parameters such as data widths, memory sizes and timing delays. Control circuitry may also be parameterized, allowing scalability of the design. Parameters may also be set to allow the same primary code to be optimized for varying target technologies from ASIC libraries to different FPGA implementations.

Many companies offer IP products based around DSP solutions, that is, where the IP code is embedded onto DSP processors. This offers full flexibility, but with the obvious reduction in performance in terms of area, power and speed. Texas Instruments (TI) and ARM are two examples of extremely successful companies supplying both the chip sets and the supporting libraries of embedded components. In a similar manner, a wealth of companies delivering firm and soft IP cores has been established. This has been of particularly true for the advent of FPGA companies that not only sell the chips on which to implement the user's designs, but can also provide many of the fundamental building blocks needed to create these designs. The availability of such varied libraries of functions and the blank canvas of the FPGA brings great power to even the smallest design team. They no longer have to rely on internal experts in certain areas, allowing them to concentrate on the overall design, with the confidence that the cores provided by the FPGA vendors have been tested through use by previous companies.

A list of some current IP vendors (Davis 2006) follows:

4i2i Communications Ltd: IP Cores in Verilog and VHDL for FPGAs and ASICs. Specializing in video coding and DSP technology (http://www.4i2i.com/)
Conexant, Amphion IP Cores: IP Cores in Verilog and VHDL for FPGAs and ASICs. The company specializes in video, imaging, security, speech and audio, broadband wireless and DSP technology (http://www.conexant.com/products).
Axeon Ltd: Vindax IP Cores – Synthesizable and scalable microprocessor architecture, optimized to support machine learning, (http://www.axeon.com)
ARC: configurable cores: CPUs/DSP (http://www.arc.com/)
Digital Core Design: VHDL and Verilog cores for microcontrollers, bus interfaces and arithmetic co-processors (http://www.dcd.com.pl/)

This gives some idea of the diversity of IP products, and the increasing scale of the components. The large cores tend to come in areas where dedicated standards have indicated a desired or restricted performance, such as in the case of video coding (JPEG, MPEG) and bus and memory interfaces.

Within hard IP cores, components can be further defined (Chiang *et al*. 2001), although the definitions could be applied across all variations of IP cores, with the pre-silicon stage relating more to the soft IP core and the production stage relating to a pre-implemented fixed design hard IP core:

Pre-silicon: given a one-star rating if design verified through simulation
Foundry verified: given a three-star rating if verified on a particular process
Production: given a five-star rating if the core is production proven

When developing IP, vendors often offer low-cost deals to attract system designers to use their new product and prove its success. Once silicon is proven, the product offers a market edge to competitive products.

10.3 Evolution of IP Cores

As technology has advanced, the complexity of the granularity of the core building blocks has increased, raising the level of design abstraction. This has led to a design evolution resulting in a hierarchy of fundamental building blocks. This section gives a summary of this evolution.

Within the realms of ASICs, families of libraries evolved, bringing a high level of granularity to synthesis. At the lowest level, the libraries defined gated functions and registers. With increased granularity, qualified functional blocks were available within the libraries for functions such as UARTs, Ethernet, USBs controllers, etc. Meanwhile within the DSP domain, processors companies such as Texas Instruments Inc. (TI) were successfully producing software solutions for implementation on their own devices. It was with the development of families of arithmetic functions that the role of IP cores in design for reuse for ASIC and FPGA designs, came into play. It was a natural progression from the basic building blocks that supported ASIC synthesis. The wealth of dedicated research into complex and efficient ways for performing some of the most fundamental arithmetic operations lent itself to the design of highly sophisticated IP cores operating with appealing performance criteria.

Figure 10.2 gives an illustration of this evolution of IP cores and how they have increased in complexity with lower-level blocks forming key components for the higher levels of abstraction. The arithmetic components block shows a number of key mathematical operations, such as addition, multiplication and division, solved and implemented using a number of techniques such as carry-look-ahead and high radix forms of arithmetic. Many of the techniques to achieve this level of functionality were covered in Chapter 3.

With greater chip complexity on the horizon, the arithmetic components became building blocks for the next level of complexity hierarchy, such as filter banks which consist of a large array of MAC blocks. This led to the development of fundamental DSP functions such as FFTs and DCTs. These examples are matrix-based operations, consisting of a large number of repetitive calculations that are performed poorly in software. They may be built up from a number of key building blocks based on multiply and accumulate operations. The structured nature of the algorithms lends itself to scalability, allowing just a small number of parameters to control the resulting architecture for the design. Such examples of parameters could be the choice of wordlength and truncation. Other examples would be based on the dimensions of the matrix operations, relating, for example, to the number of taps on a filter. This allows the work devoted to a single application to be expanded to meet the needs of a range of applications.

Other more complicated foundation blocks were developed from the basic arithmetic functions. More complicated filter-based examples followed such as adaptive filters implemented by the LMS algorithm or the more complex QR based RLS algorithm. The latter is given as an example in Chapter 12. Highly mathematical operations lend themselves well to IP core design. Other examples, such as forward error correction chains and encryption, whereby there is a highly convoluted manipulation of values, have also been immensely successful.

IP cores have now matured to the level of being able to perform full functions that might have previously been implemented on a number of independent logic blocks or devices. Again, this raises the level of complexity within the base logic blocks. An example of this is the DCT. The development of a high-performance IP block for the DCT has been an area of keen interest (Hunter 1999). A complex component in itself, it is a fundamental part of the JPEG and MPEG image compression algorithms which are themselves commercial IP products.

Each of the levels of design abstraction is covered in more detail over the following sections.

10.3.1 Arithmetic Libraries

The examples in Figure 10.2 list a number of basic mathematical operations namely addition, multiplication, division and square root. The efficient hardware implementation of even the most basic of these, that is addition, has driven an area of research, breaking down the operations to their lowest bit level abstraction and cleverly manipulating these operations to enhance the overall performance in terms of area, clock speed, and output latency (Ercegovac and Lang 1987, Hwang 1979, Koren 1993, Schwarz and Flynn 1993, Srinivas and Parhi 1992, Takagi *et al*. 1985). The following sections give some detail regarding the choice of arithmetic components and how parameters could be included within the code.

Fixed-point and Floating-point Arithmetic

The arithmetic operations maybe performed using fixed-point or floating-point arithmetic. As discussed in Chapter 3, with fixed-point arithmetic the bit width is divided into a fixed-width magnitude component and a fixed-width fractional component. Due to the fixed bit widths, overflow and underflow detection are vital to ensuring that the resulting values are accurate. With floating-point arithmetic, the numbers are represented as shown earlier in Figure 3.2 and provide a much better dynamic range.

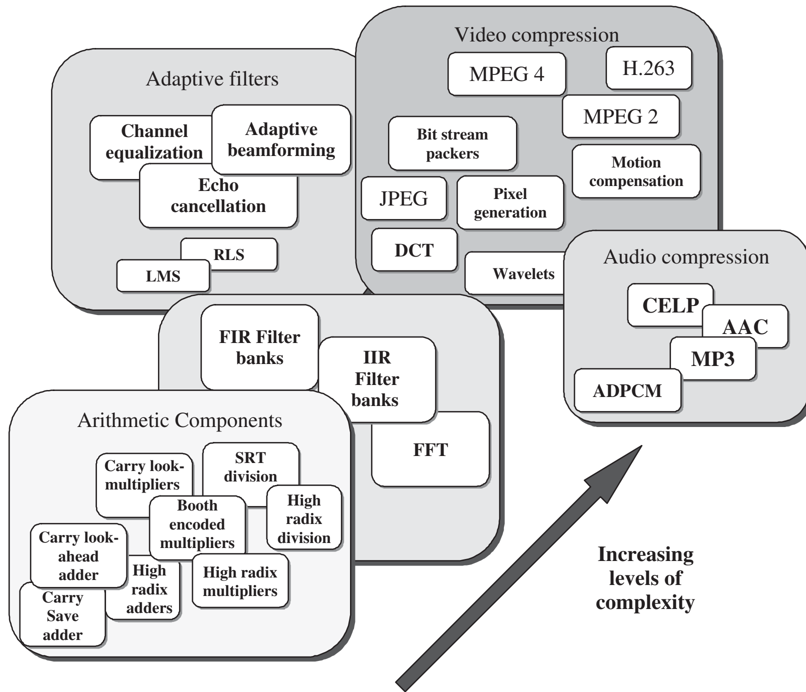


Figure 10.2 Evolution of IP cores

Although the number representation within the data width differs for fixed- and floating-point design, there is overlap on how the main functionality of the operation is performed, as illustrated for multiplication in Figure 10.3, and there has been research into automating the conversion from fixed- to floating-point, (Shi and Brodersen 2003).

As was discussed in Chapter 3, floating-point and fixed-point arithmetic have their advantages and disadvantages. A QR filter implementation on FPGA using floating-point arithmetic is given in Lightbody *et al*. (2007). Earlier studies (Walke 1997) of the QR filter implementation showed floating-point arithmetic to be a more efficient and higher-performing solution for particular application (adaptive beamforming for radar) when compared with fixed-point arithmetic. However, this investigation concentrated on ASIC implementations. With the FPGA implementation the issue of fixed- versus floating-point arithmetic was clouded and no longer clear-cut, and seems heavily dependent on the capabilities of the prospective FPGA device. Indeed, with the absence of dedicated floating-point hardware in FPGA, the performance is poor and work by Craven and Athanas (2007) suggests that the performance can be even poorer than a processor implementation.

Addition, Multiplication, Division and Square Root

There has been an extensive body of work devoted to high-performance implementations of arithmetic components, (Ercegovac and Lang 1987, Hwang 1979, Koren 1993, Schwarz and Flynn 1993, Srinivas and Parhi 1992, Takagi *et al*. 1985) some of which has been highlighted in Chapter 3. For addition and subtraction and also now for multiplication, the evolution of dedicated fixed-point additive and multiplicative hardware has negated the need for any detailed

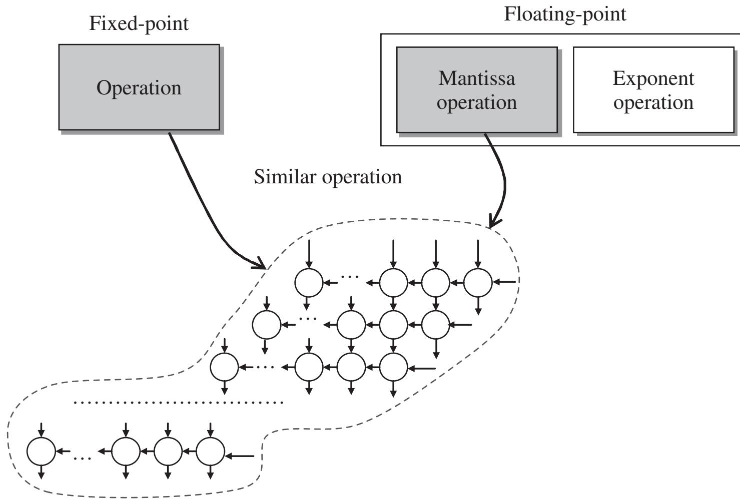


Figure 10.3 Fixed- and floating-point operations

implementation discussion and, to a great extent, dedicated cores are irrelevant, except of course, in floating-point. Floating-point arithmetic cores have been developed for both ASIC and FPGA by Amphion semiconductors (now Conexant) and also by Northeastern university (University N 2007). As Chapter 3 highlighted, there are a number of ways of performing division, including recurrence methods and division by functional iteration. These techniques lead to an architectural description with defined multiplicative and additive stages which can be scaled to meet the wordlength requirements. The key trade-off is the use of the LUT to create the initial estimate which acts to speed up the process, but which makes the core less scalable.

As the level of abstraction within VLSI design has been raised beyond the bit level arithmetic computations described above, tools such as Synplicity enable key arithmetic components of fixed-point multiplication and addition to be inferred using simple arithmetic operands. Within FPGAs, the devices have high-performance fixed-point multiplication operations already implemented on the device.

10.3.2 Fundamental DSP Functions

This section gives some examples of the more intricate cores that can be based from lower-level arithmetic modules. See Chapter 2 for more detail of the following algorithms.

FFT: the FFT is a powerful and efficient matrix based operation. It is widely used and can be utilised in applications such as orthogonal frequency division multiplexing, a powerful modulation/demodulation scheme for communications applications such as wireless (IEEE802.11a/g) or broadcasting (DVB-T/H)

DCT: this is another matrix-based operation that is of great importance for many image processing applications.

LMS and RLS filter: adaptive filtering algorithms are used in many applications such as mobile telephony and radar.

Wavelets: wavelet decomposition has a range of signal processing applications, including image compression.

Filter banks: filter bank architectures can be highly regular, consisting of a systolic array of repetitive cells, e.g. MAC operations. Within such a regular structure there is an obvious ability to create scalable code to allow a range of filter taps to be supported. The complexity increases drastically if the level of hardware needs to be kept to a minimum, with hardware reuse focused on a core number of operations on which to schedule the full functionality of the algorithm.

In each case, the aim is to derive the circuit architecture in such a way so as to preserve regularity and programmability as this will allow the core to be parameterized and produce reasonably consistent performance results across the range of parameters. Take for example the FFT; it is possible to decompose the FFT in such a way that large FFTs can be created from smaller block sizes. This is a clear means of allowing the block size to be parameterized, even though it may not be the most area- and speed-efficient solution; however, it will give good performance across the block size parameters. The availability of a scalable adder also means that performance is predictable for different wordlengths, although this becomes an issue for the multiplier function as these have limited wordlength in Xilinx Virtex (18-bit) and Altera Stratix (36-bit) FPGA families. Once these wordlengths are exceeded, the multipliers have to be constricted from several blocks.

10.3.3 Complex DSP Functions

The DSP functions summarized above form main algorithmic functions within a range of applications and would need to be scalable across key parameters. However, there are some applications where the performance is reasonably well defined within the standards such as JPEG and MPEG. As already discussed, the DCT forms a key function within JPEG compression. This is illustrated in Figure 10.4. A key balance is to create the more complex DCT functionality directly in hardware and dedicate the rest of the functionality into software. It might then be possible to reuse the hardware block to create bigger block sizes, if needed. MPEG4 video encoding involves additional motion estimation and compensation blocks which represent some 90% of the processing time. Thus, there is interest in developing parameterizable cores for motion estimation.

10.3.4 Future of IP Cores

As the level of abstraction within the core building blocks of designs increases, the role of the designer is moving more toward that of a system integrator, particularly with development using current FPGA devices enabling full system functionality on a single device. For the growth in IP core use to continue, other aspects within the design flow will need to be addressed. The following is quoted from Gajski *et al.* (2000):

"IP tool developers provide IP providers and system integrators with design methodologies and tools to support IP development and system integration".

The following section details the process from design concept to scalable (soft IP) solution.

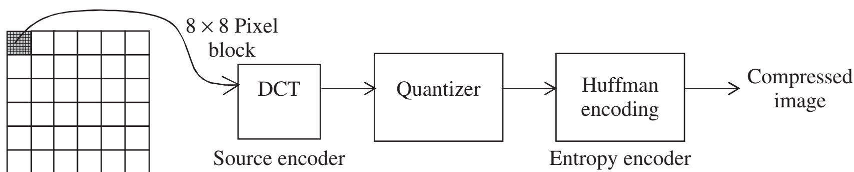


Figure 10.4 JPEG image compression

10.4 Parameterizable (Soft) IP Cores

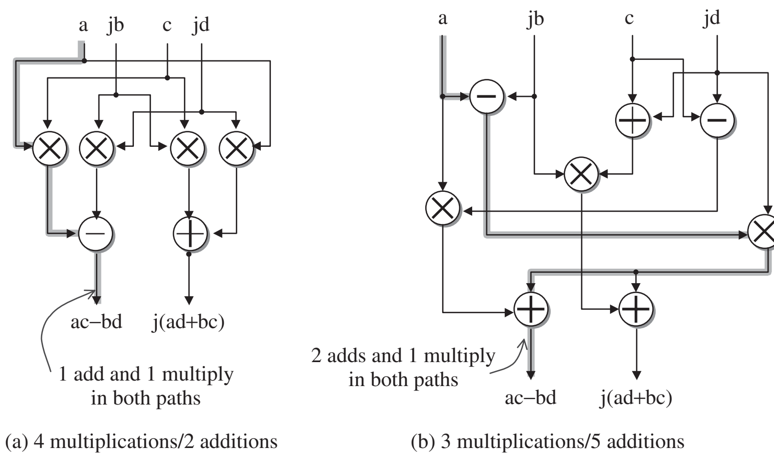
This section will cover the development of parameterizable IP cores for DSP functions. The starting point for the hardware design of a mathematical component may be from the SFG representation of the algorithm. Here, a graphical depiction of the algorithm shows the components required within the design and their interdependence. The representation could be at different levels, from the bit-level arithmetic operations through to the cell-level functions. The SFG gives information regarding the flow of data through these components and provides a very powerful representation of the design from which the hardware architecture can be developed.

An example SFG is given in Figure 10.5 which depicts two variations of how to perform complex multiplication. This simple example shows how, by reordering the arithmetic operations, changes can be made to enhance the area or critical path of the resulting hardware architecture. This solution for the complex multiplication of two numbers $(a + jb)$ and $(c + jd)$ is:

$$(a + jb)(c + jd) = [a(c - d) + d(a - b)] + j[b(c + d) + d(a - b)]$$

Figure 10.6 shows the conventional design flow for a DSP-based circuit design, starting from the SFG representation of the algorithm. Typically, this cycle would be re-iterated, either because of design changes or lack of desired performance. Many of the steps were covered in the previous chapters. The process starts with a definition of the algorithm in terms of the SFG or DFG description. A detailed analysis is then carried out as highlighted in Chapter 2, to determine key aspects such as internal wordlengths, truncation issues, throughput rate requirements. These then form the bounds for the creation of the circuit architecture which was covered in detail in Chapter 8. When complete, the circuit architecture is coded up as a HDL-based description and conventional synthesis tools used to create the final design. If a certain aspect of the specification were to be changed, such as wordlength, then the traditional full design flow would need to be repeated.

The development of the IP core where the HDL is parameterized allows this flow to be dramatically altered, as shown in Figure 10.7. The design needs to encompass the initial studies in the effects of wordlength and arithmetic on SNR, area and timing performance. Effort would be



(a) 4 multiplications/2 additions (b) 3 multiplications/5 additions

Figure 10.5 Complex multiplication SFG

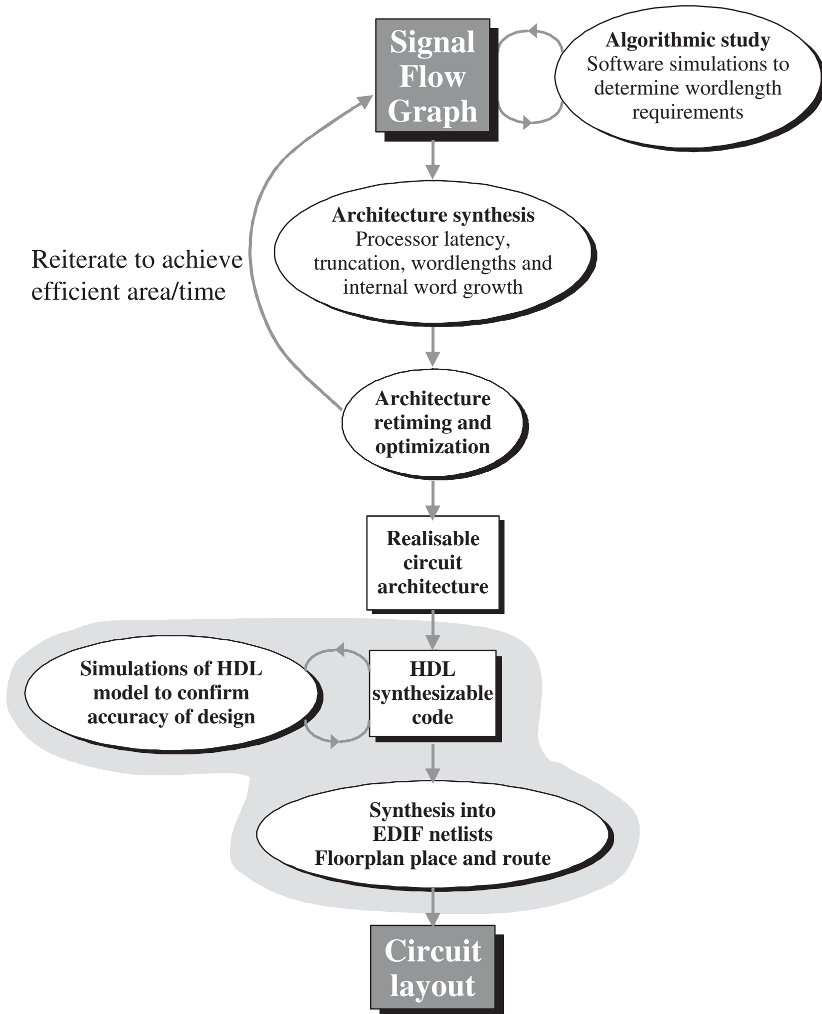


Figure 10.6 VLSI circuit design flow

needed to ensure that operation scheduling would still be accurate if additional pipeline stages were to be included, the aim being that the parameterization of the core would lead seamlessly to a library of accurate implementations targeted to a range of specifications, without the need to alter the internal workings of the code. The system should effectively allow a number of parameters to be fed into the top level of the code. These would then be passed down through the different levels of abstraction of the code to the lowest levels. Obviously, considerable effort is needed at the architecture level to develop this parameterizable circuit architecture. This initial expense in terms of time and effort undoubtedly hinders the expanded use of design for reuse principles. However, with this initial outlay, great savings in company resources of time and money may be obtained. The choice of which design components on which to base further designs and develop them as

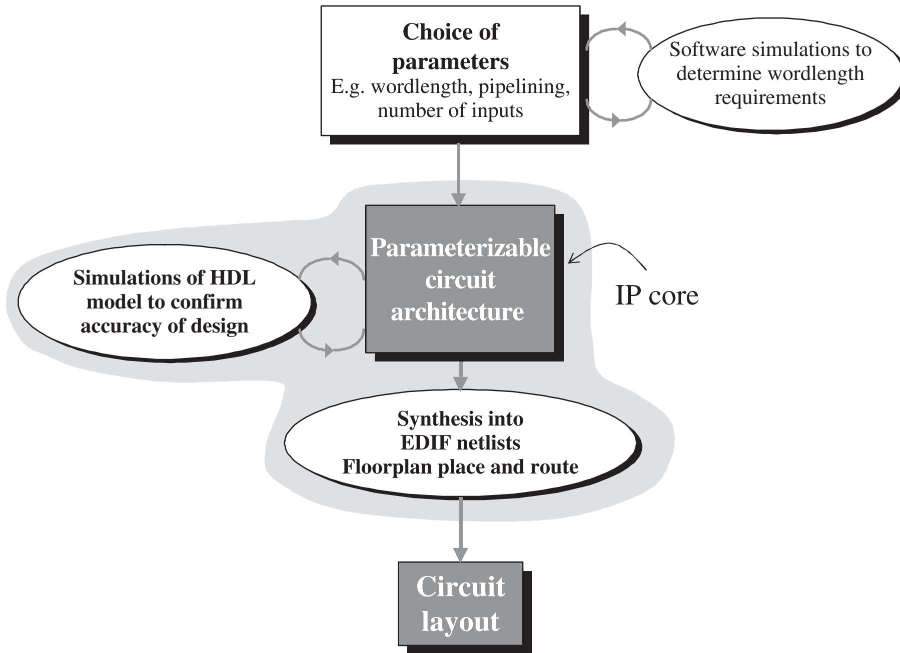


Figure 10.7 Rapid design flow

IP, is vital for this success. The initial expenditure must in the long run result in a saving of resources.

Future design engineers need to be taught how to encompass a full design for reuse methodology from the project outset to its close. The design process needs to consider issues such as wordlength effects, hardware mapping, latency and other timing issues before the HDL model of the circuit can be generated. The aspects that need to be considered create a whole new dimension to the design process, and designers need to keep in mind reusability of whatever they produce whether for development or test purposes. If a design is developed in a parameterized fashion then initial analysis stages can be eliminated from the design flow, as illustrated in Figure 10.7, allowing additional circuits to be developed and floorplanned in extremely short time scales, typically in days as opposed to months. This activity represents a clear market for IP core developers (Howes 1998) as it can considerably accelerate the design flow for their customers. However, it requires a different design approach on behalf of the IP core companies to develop designs that are parameterizable and which will deliver a quality solution across a range of applications.

10.4.1 Identifying Design Components Suitable for Development as IP

Within a company structure, it is vital that the roadmap is considered within the development of IP libraries, as there is a greater initial overhead when introducing design for reuse concepts. Greater success can be obtained by taking an objective look into possible future applications, so that a pipeline of developments can evolve from the initial groundwork, thereby justifying incorporation of design for reuse from the outset. It is often possible to develop a family of products from the

same seed design, by including parameterization in terms of wordlength and level of pipelining, and by allowing scalability of memory resources and inputs.

Larger designs may need to be broken down into manageable sections that will form the reusable components. This is particularly true for large designs such as MPEG video compression, whereby a range of different applications would require slightly different implementations and capabilities. By picking out the key components that remain unchanged throughout the different MPEG profiles and using these as the key hardware accelerators for all of the designs, vast improvements in time-to-market can be met. Furthermore, existing blocks from previous implementations also have the advantage of having been tested in full, especially if they have gone to fabrication or deployment on FPGA. Reusing such blocks adds confidence to the overall design. Existing IP may also form the key building blocks for higher-level IP design, creating a hierarchical design.

Figure 10.8 illustrates some of these key points. A depiction is given of repetitive blocks being used in multiple designs. Similarly, the figure shows larger designs built up from a hierarchy of lower-level blocks. The company roadmap is illustrated, highlighting the importance of careful choice of which components to assign the extra design effort. Scalability is another key consideration. Can the design be expanded and scaled to meet the demands of a variation of applications? An example of this is in speech compression whereby a codec may need to work on a range of channel numbers depending on the overall application, for example, a simple DECT phone or for a telecommunications network.

10.4.2 Identifying Parameters for IP Cores

Identifying the key parameters when developing an IP core requires a detailed understanding of the range of implementations in which the core may be used. The aim is not to just create as much flexibility into the design as possible, but to identify the benefits that this additional effort will bring in the long run. *Over-parameterization* of a design not only affects the development time, but also affects the verification and testing to ensure that all permutations of the core have been considered. In other words, consider the impact on the design time and design performance of adding an additional variable and weigh this against how the added flexibility will broaden the scope of the IP core!

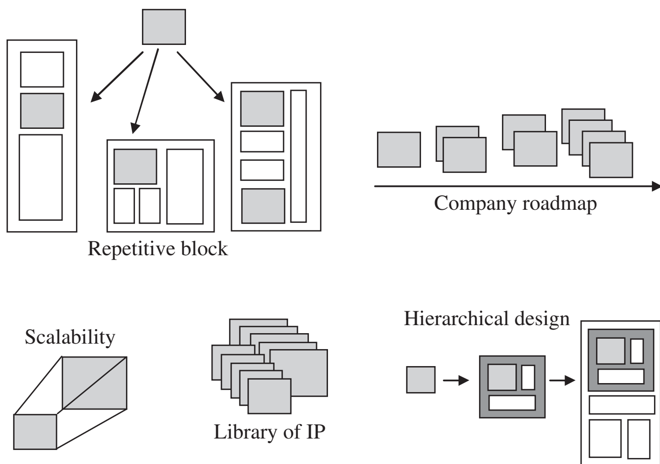


Figure 10.8 Components suitable for IP

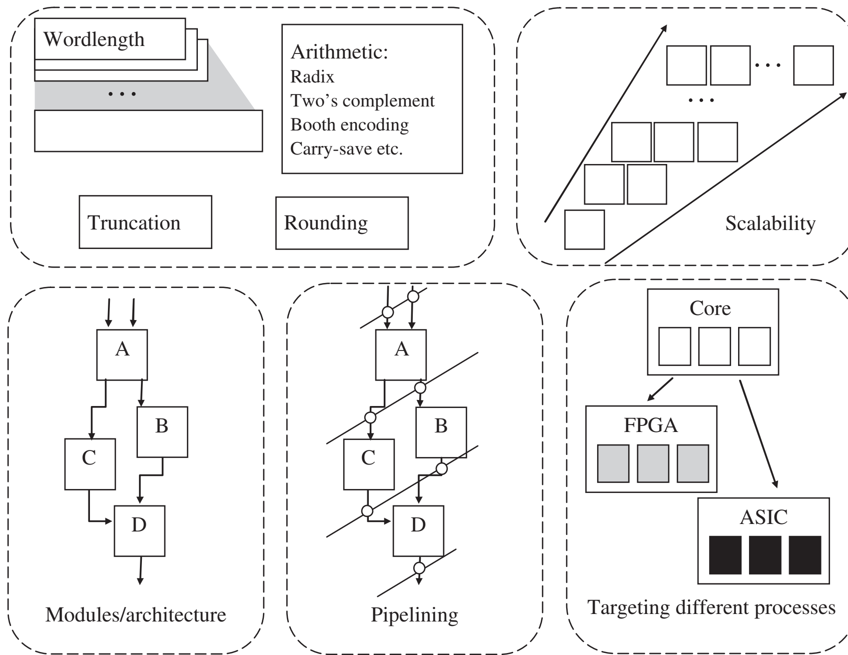


Figure 10.9 IP parameters

There follows a list of example parameters, some of which are illustrated in Figure 10.9:

- modules/architecture
- wordlength
- memory
- pipelining
- control circuitry
- test environment

An obvious parameter is wordlength. Choice of wordlength comes down to a trade-off between SNR and performance criteria such as area and critical path. Figure 10.10 gives an illustration of such an analysis, plotting the SNR against a range of wordlengths. From this simple example it can be seen that increasing the wordlength further will not significantly improve the overall performance. For some components such as addition, an increase of one bit will linearly scale the area of the resulting implementation. However, for components such as multiplication and division, increasing the wordlength will have an exponential effect on the area. For this reason, wordlength analysis is critical as well as truncation or rounding analysis.

Enabling full scalability of the hardware is another example. The diagram shows a single block that is repeated as the needs of the application scale. An example of this could be for an N-tap filter. As N increases the hardware should be able to increase the logic blocks accordingly to meet the application requirements.

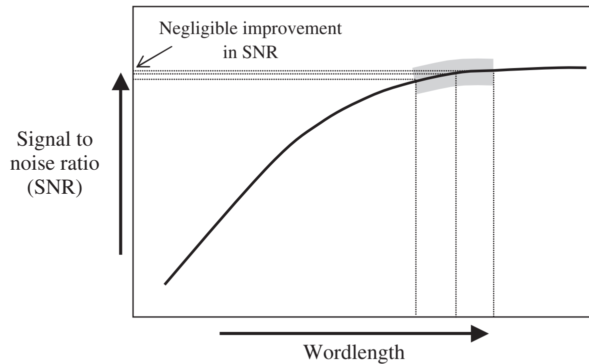


Figure 10.10 Wordlength analysis

Memory may also need to be scalable to account for the different wordlengths, but also for variations in the number of inputs or stored values. More importantly, how the memory architecture is created is also very important. This was highlighted in Chapter 5 where Tables 5.3 and 5.8 give the list of memory available in Altera Stratix and Xilinx Virtex FPGA families respectively. It is clear that embedded RAM gives area efficient, coarse memory blocks, but embedded LUTs allow faster operation due to the highly parallel nature of their operation and the fact that they can be co-located to the hardware. This would need to be taken into consideration when the core is developed.

Different applications could be implemented using a combination of sub-blocks. This is represented in the Figure 10.9 with the module blocks labelled A, B, C and D, depicting a simple architecture. As with area, additional bits within the wordlength will have a knock-on effect on the critical path, possibly resulting in the need to introduce further pipeline stages throughout the design, as illustrated, using the same simple architecture of sub-blocks, A, B, C and D. Enabling the level of pipelining to be varied is an important parameterizable feature that can widen the application opportunities.

Allowing flexibility in the data rate and associated clock rate performance for an application will also require the ability to vary the level of pipelining within the design to meet the critical path requirements. As already mentioned, this may tie in with changes in the wordlength and often a variation in the number of pipeline stages will be part of the reusable arithmetic cores. Obviously an increase or decrease in the number of pipeline cuts in one module will have a knock-on effect on the scheduling and timing for the associated modules and the higher level of hierarchy. For this reason, control over the pipeline cuts within the lower-level components of a design must be accessible from the higher level of the module design, so that lower-level code will not need to be edited manually.

There may also be a need to develop scalable control circuitry that will allow for the changes in parameters, such as additional pipelining delays or any increase in the number of inputs or wordlength, that may have an effect on the scheduling and timing of the module. One important example is how to develop parameterizable control to handle the scheduling of multiple operations onto a single instantiation of hardware.

Obtaining a fully parameterizable design is a convoluted issue that needs to be planned; the overall goal is that the code can be resynthesized for a new architecture in extremely short timescales

while still meeting the desired performance criteria. This a key factor for the success of an IP core. It is crucial that the resulting core has performance figures in terms of area, power and speed, comparable to a handcrafted design. As usual, the process comes down to a balance between time and money resources and the performance criteria of the core.

An alternative method for developing parameterizable cores can be to create a software code to automate the scripting of the HDL version of the module. This is particularly useful with Verilog as it does not possess the same flexibility in producing scalable designs as VHDL.

Consideration must be made to the level of parameterization within a design. From the outset it would seem reasonable to include as much flexibility within a design as possible. Surely doing this would widen the market potential for the IP core. Gajski *et al*. (2000), highlight the issue of *over-parameterization*. The more variables that are placed within a core the harder it is to verify the full functionality of each permutation of the design. There is also the aspect that a design that has been made overly generic may not command the performance requirements for a specific application. Gajski stated that, as the number of parameters increases, there is a decrease in the quality and characterization of the design, that is, how well the design meets the needs of the user. There are also the added complications with verification and testing. These points are highlighted in Figure 10.11. It is essential to consider such issues when choosing parameters, so as to find a balance between design flexibility and design reliability.

10.4.3 Development of Parameterizable Features Targeted to FPGA Technology

Many of the IP designs targeted for ASIC implementation can be expanded for FPGA implementations and *vice versa*. Each technology has its own characteristics, but by accounting for these differences, it becomes viable to create the code in such as way that cores can be retargeted at the top level to the FPGA or ASIC family of choice. This is of particular importance as FPGAs are rapidly progressing, thus legacy code needs to accommodate additions for future devices and packages. This was illustrated in the example in Figure 10.9 where it is shown how one seed design can be used as a base from which to generate technology-specific designs targeted to ASIC and FPGA.

Allowing the same code to be altered between the two technologies has the obvious advantage of code reuse, broadening the market for the product by not limiting the target technology. However, it also allows for a verification framework, whereby cores are prototyped on FPGA; the same code is then retargeted to ASIC. There is obviously no guarantee that the code conversion from FPGA to ASIC implementations will not in itself incur errors. However, the ability to verify the code on a real-time FPGA platform brings great confidence to the design process and enables even the functional design to be enhanced to better meet the needs of the specification. Typically the

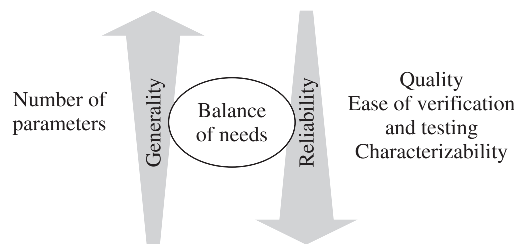


Figure 10.11 Effect of generalization on design reliability (Gajski *et al*. 2000)

following issues need to be considered when switching between the two main streams of target implementation:

FPGA embedded arithmetic
ASIC custom designed arithmetic cores
Technology specific memory
Pipelining may need to be adjusted to meet the desired performance on the chosen technology.
Area and resources. The chosen FPGA will have limited resources compared with ASIC. The designer may wish to keep within a particular grade and size of FPGA device for cost reasons. This may impact the maximum achievable clock rate while limiting the amount of possible pipelining.

IP components for FPGA implementation offer a great acceleration of the development full systems on a single device. Furthermore, vendors such as Xilinx and Altera have collaborated with external design houses to develop a wide library of functions targeted to their devices, available to download from their home websites, specifically the Xilinx Alliance program and Altera's Megafunctions Partners program. The accessibility of such material enhances the user's capabilities when using their FPGA devices and expands the market sector. It is a symbiotic relationship, allowing the IP core vendors, the FPGA companies, and design houses to flourish.

Memory Block Instantiation

One example of the variations between FPGA devices are the memory blocks. Each family has its own architecture for these blocks. In Verilog for FPGA implementation, one of two solutions can be used. Instantiations of block RAMs for the target device can be scripted with DEFINEs at the top level of the code pointing to the memory of choice. Alternatively, the code can be written in such a way as to 'infer' the application of a memory, which will be picked up during synthesis by the modern tools and they will instantiate the memories accordingly. However, slight improvements may be still be obtained if the memory instantiations are handcrafted, but this results in more complex code.

Arithmetic Block Instantiation

Different target FPGAs may have variants of arithmetic operations available for the user. The implementation of these embedded blocks could be inferred within the code. However, some customization may be required to build up arithmetic operations on wider bit widths. Another example could be the implementation of floating-point arithmetic blocks using these embedded FPGA modules as the core computation blocks. If the code is to be employed over a range of FPGA families, and even between FPGA and ASIC, then there needs to be a facility to define the operator choice at the top level. Within Verilog, this would be done through the use of DEFINEs held in a top-level file, allowing the user to tailor the design to their current requirements, providing them with three choices:

1. Infer arithmetic operation: $A + B$, etc.
2. Instantiate built-in operators available on FPGA device
3. Instantiate custom crafted arithmetic operators

The later may be the better choice for multiplication of larger numbers or floating-point arithmetic. Or it may also be important to use custom arithmetic functions if extra pipelining stages

are required within the operators to meet critical path demands. This is design dependent and may require analysis.

Parameterized Design and Test Environment

All associated testing code accompanying the IP core should be designed with scalability in mind. Bit-accurate software models used for functional verification should have the capability to vary bit widths to match the IP core. For cycle-accurate testing, timing must also be considered. Test benches and test data derivation are also required to be parameterizable, allowing for a fully automation generation of an IP core and its associated test hardness. The use of software such as C to generate the test harness and test data files may be advantageous in the development of the IP core, and is illustrated in Figure 10.12.

10.4.4 Application to a Simple FIR Filter

This section gives an example of parametric design applied to a simple FIR filter given originally in Figure 2.13. The key parameters for the design will be highlighted and suggestions as to how they can be incorporated into the code will be made. A more complete FIR design example is given in Chapter 8.

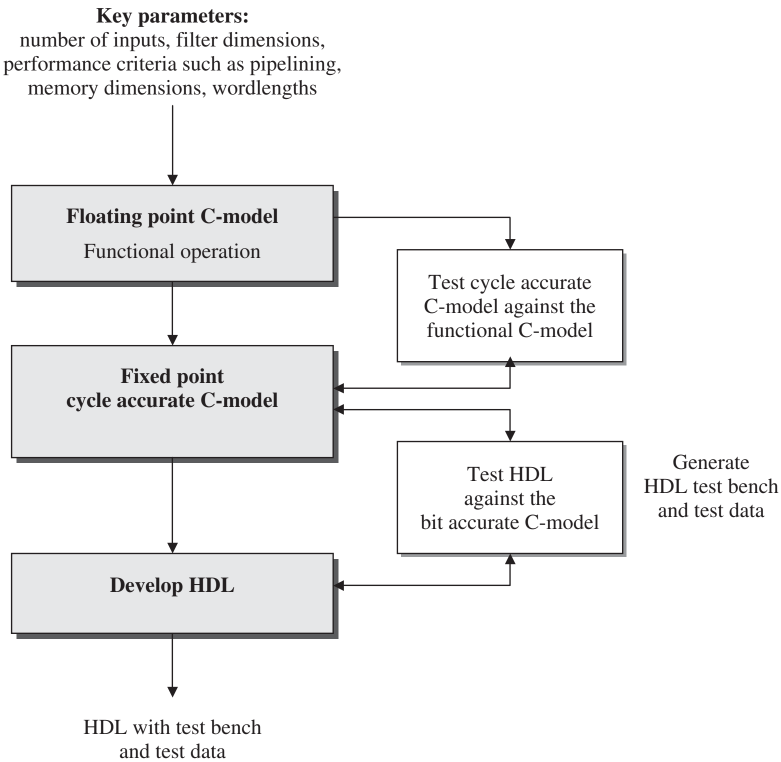


Figure 10.12 Test design flow

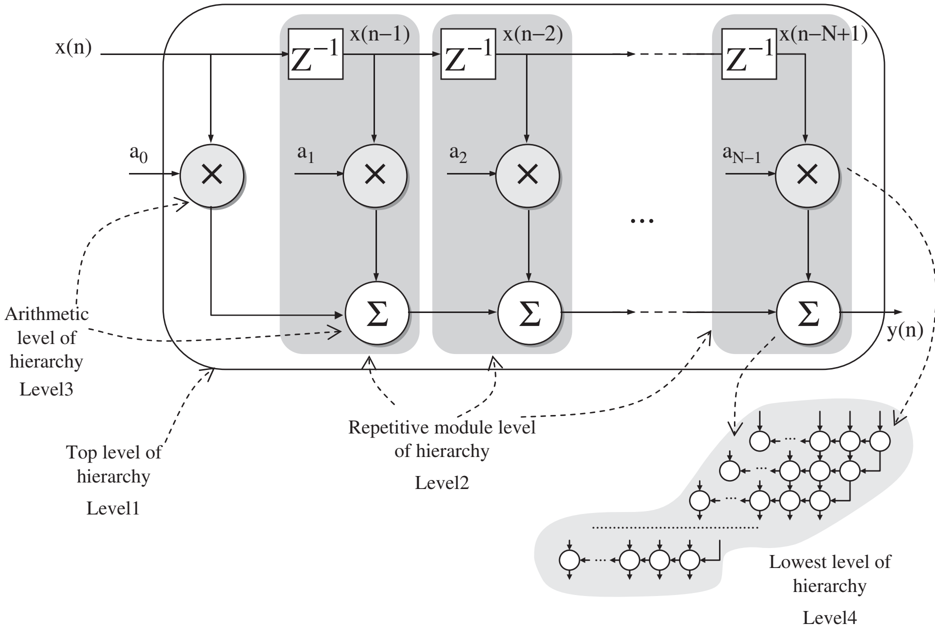


Figure 10.13 FIR example

In this example the FIR filter has four levels of hierarchy, as depicted in Figure 10.13, with the individual arithmetic units being the lowest level of hierarchy and the full FIR filter being the highest level of hierarchy:

Level 1: This is the top level of the filter structure within input $x(n)$ and output $y(n)$.
Level 2: The top level of the FIR filter can be composed of a single multiplier for $a_0x(n)$ followed by a number of delay–MAC modules (shown in the shaded boxes in Figure 10.13).
Level 3: This is the arithmetic operation level, consisting of the multiply, add and delay modules.
Level 4: The arithmetic modules can be broken down to another lower level of hierarchy, performing the bit-level operations which will usually not be relevant for FPGAs unless the designer is building multipliers from adders and LUTs.

Another dimension of the design may be the folding of the FIR operations onto a reduced architecture, that is, the hardware modules are reused for different operations within the filter, as depicted in Figure 10.14. In this example, all the MAC operations are performed on one set of multiplier and adder modules. Multiplexers are used to control the flow of data from the output of the MAC operations and back into the arithmetic blocks. The scheduling of operations onto a single unit provides technical challenges that are beyond the scope of this chapter, but were covered in detail in Chapters 8 and 9.

The choice of level of hardware reduction will depend on the performance requirements for the application. Taking this same example, we can provide figures to give a performance comparison

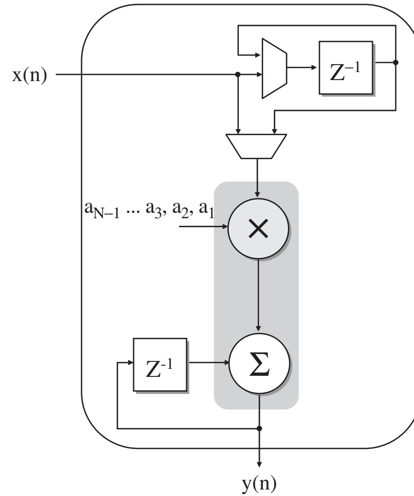


Figure 10.14 FIR example with hardware reduction

between a version without hardware reduction with our example of just a single MAC unit. Figure 10.15 depicts a 6-tap FIR example, specified by the equation:

$$y(n) = a_0x(n) + a_1x(n-1) + a_2x(n-2) + a_3x(n-3) + a_4x(n-4) + a_5x(n-5)$$

For the purposes of this example, the latencies for the multiplication and addition modules within the MAC units are 2 and 1 clock cycles respectively. These latencies are due to pipeline cuts placed within the modules and achievable with FPGA implementations. The pipeline cuts within the arithmetic modules are represented by the diagonal lines in Figure 10.15. Additional

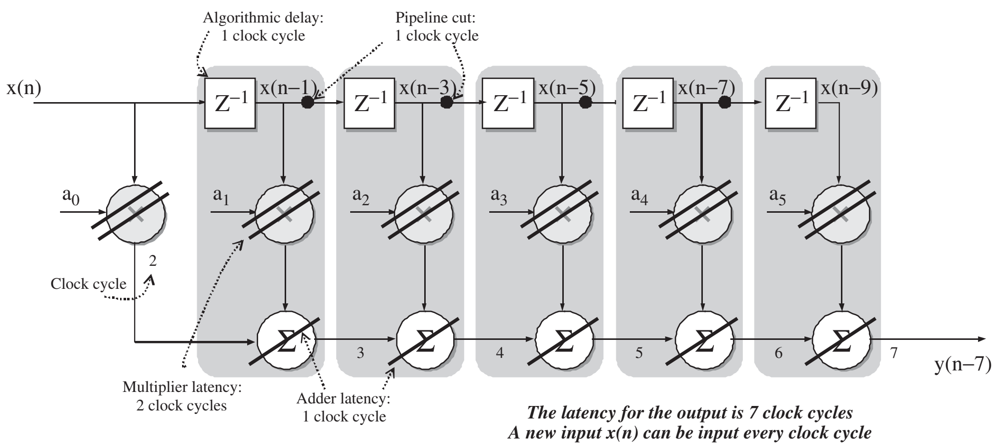


Figure 10.15 Six-tap FIR example with timing

pipeline cuts are labelled and are used to retime the circuit and schedule the operations correctly due to the latency within the arithmetic modules.

Numbers within the figure represent the clock cycle at which the operation has been performed. For example, after two clock cycles, due to the latency of the multiplier, $a_0x(n)$ has been calculated. After three clock cycles, due to the algorithmic delay plus the multiplier latency, $a_1x(n-1)$ has been calculated. These values need to be summed. However, to match the scheduling of the results for $a_0x(n)$ and $a_1x(n-1)$, $a_0x(n)$ needs to be delayed further by one clock cycle. This was covered in detail in Chapter 8. The overall latency for resulting output, $y(n)$ is seven clock cycles. In other words, the result for $y(n)$ is ready for output seven clock cycles after $x(n)$ is input into the filter, with a new result available with each subsequent clock cycle. Once again, the FIR operations can be mapped down onto one MAC unit with a few multiplexers. This means that all the operations need to scheduled appropriately so that they are performed at the correct time with the correct inputs. The latency of the arithmetic units is the same. This was covered by section 8.6.2. As always, choosing an appropriate architecture comes down to a balance of performance needs, and, area and timing criteria.

10.5 IP Core Integration

One of the key challenges of successful design reuse is with the integration of the IP cores within a user's system design. This can often be the stumbling block within a development. Investigations have been performed to highlight these issues, (Chiang *et al*. 2001, Gajski *et al*. 2000, Moretti 2001) while others have set out guidelines to try to standardize this process (Birnbaum 2001, Birnbaum and Sachs 1999, Coussy *et al*. 2002, 2006, 2007).

In order to achieve successful integration of an IP core into a current design project certain design strategies must be employed to make the process as smooth as possible. Section 10.5.1 highlights some of the pitfalls that might be met, and provides some guidance when dealing with IP cores sourced externally from the design team, whether internal or external to the company.

One of the considerations that may need to be addressed is whether to source IP components from external sources, or from within a company or organization, but from different departments. Successful intra-company use of IP requires adequate libraries and code management structures. Incorporating IP components from other design teams, whether they are inter- or intra-company, can often be the main barrier slowing down the employment of design for reuse strategies in system-level design. This is largely due to the requirement to fully validate the core across the range of functionality.

10.5.1 Design Issues

Greater success can be obtained by taking an objective look into possible future applications so that a pipeline of developments can evolve from the initial groundwork. If design for reuse is incorporated from the initial outset then there can be incredible benefits in the development of a library of functions from the initial design. To summarize:

- Need to determine the parts of the design that will be useful in future developments.
- What are the possible future applications?
- Study the road map for future products.
- Is there a possibility of developing a family of products from the same seed design?
- How can a larger design be partitioned into manageable reusable sections?
- Find existing level of granularity, i.e. is there any previous IP available that could provide a starting level for development?

Outsourcing IP

One of the limiting factors of using outsourced IP is the lack of confidence in that IP. The IP can be thought of as having different grades, with a one-star relating to a core verified by simulation, a three-star would relate to a core that has been verified through simulation on the technology, that is, a gate level simulation. Finally, a five-star IP core provides the most confidence as it has been verified through implementation (Chiang *et al*. 2001).

FPGA vendors have collaborated with the IP design houses to provide a library of functions for implementation on their devices, and this brings a level of confidence to the user. The aspect of core reliability is not as crucial for FPGA as it is for ASIC, but it is still important. Time wasted on integration issues of the IP into the user's product may be critical to the success of the project.

Certain questions could be answered to help determine the reliability of an IP vendor:

- Has the core been used in previous implementations for other users?
- Do the company supply user guide and data book documentation?
- Does the core come supplied with its own testbench and sufficient and suitable test data?
- Will the company supply support with the integration, and will this incur an added cost?

In-house IP

For a small company within the same location, it would be a much easier task to share and distribute internal IP. However, this task becomes logistically difficult with larger companies spanning a number of locations, some of which may be in different time zones as well as the physical distance. Furthermore, there is always competition of staff time resources. Project leaders may not be happy with their team spending time supporting the integration of their legacy code into another group's development. This time resource needs to be recognized at a company level and measures put in place to support such collaboration.

It would be appropriate for the company to introduce a structure for IP core design, and give guidelines on the top-level design format. Stipulating a standard format for the IP cores could be worthwhile and can create greater ease of integration. Forming a central repository for the cores once they have been verified to an acceptable level would be a necessity to enable the company's full access to the IP. Most companies already employ some method of code management to protect their products.

10.5.2 Interface Standardization and Quality Control Metrics

A major limiting factor in the use of IP has been due to integration problems. Another issue is the lack of confidence that a developer might have in third party IP. This section gives a brief summary of some of the standards

Virtual Socket Interface Alliance

The Virtual Socket Interface Alliance (VSIA) is a standards organization that was established in 1996 to support the standardization and adoption of IP core development and design for reuse practices in the electronics industry. It was an alliance that was representative of a cross-section of the SoC industry. After 11 years, this organization has recently ceased operation within the SoC industry. During this time in operation, VSIA established more than 20 standards, specifications and technical documents distributed to members at a small fee, or no fee at all. They have passed on their standards and work to be handled and progressed by other organizations who develop IP and electronics standards. One such example is the IEEE.

One of their particular successes has been the VSIA Quality IP (QIP) metric (VSIA 2007). This free document covers the major design considerations met when deciding to integrate third party IP. It provides a metric for vendor qualification as well as metrics for the comparison of soft, hard and verification IP. It is well established and has been widely employed by developers over several years since publication.

In addition to supporting documentation the QIP is implemented by an Excel spreadsheet in which the user has to insert certain information and parameters about the product and vendors that they are considering to use. The spreadsheet consists of a number of individual sheets, one for vendor assessment, and several for soft IP integration, IP development among others. The list below gives some of the questions asked about the vendor:

Processes:
Is the development process for IP defined and documented?
Does a process for measuring customer satisfaction exist and is this process used consistently?
Verification:
Is the requirements change process for IP defined, documented and followed consistently?
Does a detailed and coordinated test plan exist and can this document be made available to the user?
Has the IP been used in a real production environment?
Revision control:
Are the revision control scheme and related guidelines fully documented?
Are end-of-life notices for IP given well in advance (at least 6 months)?
Distribution:
Does the user get notified automatically if supporting deliverables of the IP change, or if new features or revisions of the IP become available?
Consistency:
Are the IP and associated deliverables provided in a standardized consistent manner?
Support:
Is a problem-reporting infrastructure and corresponding processes in use?
Can IP be evaluated before purchase?
Documentation:
Does the documentation cover all aspects necessary to successfully integrate the IP into a system?
Vendor confidence:
Has the company been in existence for over 5 years?
Are there more than 50 employees?
Can customer references be made available?

This is just a subset of the detail asked within the QIP spreadsheet. The sheet for the metric for Soft IP integration asks questions such as:

- Has the IP be integrated by a team other than the developers?
- Is there training?
- Has it been in FPGA or IC production?
- Is there adequate documentation?
  Interface information; instantiation guidance; area and power estimates on the relevant technology; verification methods; deliverable list.
- Integration:
  Build environment: are scripts supplied? Scan insertion scripts? Portability with other design tools?

This list is far from complete. The QIP asks questions in great detail.

The Soft IP development sheet is a useful guide to those planning to develop an IP core. It can act as a checklist for providing quality code and represents much of what has been discussed in this chapter. These are some of the most relevant comments:

- Are all delay values passed as parameters rather than being hard coded?
- Are values such as bit widths, register addresses, etc. represented by parameters?
- Is there consistency with signal naming standards?
- Verilog: are the 'define' references kept to one file?
- VHDL: are the deferred constants obtaining their values from one package?
- Does functional coverage receive 98% for all synthesizable statements?
- Are the regression tests scripted with log files?

The QIP documentation is a very useful tool and as a result has been widely used. It is constantly going through revisions and the VSIA has now passed it on to the IEEE where it will be developed further.

VSIA also had an IP protection working group and had been currently involved in developing encryption standards for IP cores in addition to their existing soft and hard tagging standards. Commercial entities ChipEstimate (www.chipestimate.com) and Design and Reuse (www.design-reuse.com), benefited from the use of VSIA's IP transport specifications.

10.6 ADPCM IP Core Example

Adaptive differential pulse code modulation (ADPCM) is one of a number of speech compression algorithms defined by the International Telecommunication Union (ITU). It is a popular choice for speech compression for a number of applications, from digital cordless phones to telephone networks. It provides the compression of a 64 kbit/s pulse code modulation (PCM) audio channel to a 40, 32, 24 or 16 kbit/s ADPCM audio channel and *vice versa*. The standard operating rate of 32 kbit/s offers high-quality compressed speech with no noticeable loss in fidelity in comparison with PCM. Table 10.2 gives a summary of the key ITU standards for PCM and ADPCM.

Due to the variability in the range of applications within which ADPCM could be used, developing a design to match the diverse needs is a challenge. Figure 10.16 gives the basic structure for APDCM compression from a PCM input. The crux of APDCM compression is adaptive prediction of the current sample using the previous speech samples. Once calculated the predicted sample is subtracted from the actual current sample. Only this error signal is quantized and encoded for transmission. Then on the decoding side, that is, the receiver, the reverse process is performed to regenerate the actual speech sample.

This technique relies on the fact that speech offers some level of pseudo stationarity. That is, over a short period of time, about 20 ms, the statistics of the signal remain largely unchanged. This allows for the adaptive prediction used in ADPCM. For the other variants of the ADPCM ITU (G.726) standard there are different quantizer tables to perform the encoding to 40, 32, 24 or 16 kbit/s, and similarly, with the decoding. Or alternatively, a single quantization table can be used for all four compression rates (ITU G.727), but with the quantization tables for the 16, 24, and 32 kbit/s data rates being subsets of the table for the 40 kbit/s data rate. However, this optimization is at a cost of a slight degradation to the signal-to-quantization-noise ratio.

As it can be seen in the figures for encoding (Figure 10.16) and decoding (Figure 10.17), there are certain components that are consistent in both and could be used to perform both tasks, therefore

Table 10.2 Summary of key ITU standards for PCM and ADPCM

Compression standard	Rates (kbit/s)	Features
PCM (G.711)	64	Conversion from 14-bit or 13-bit uniform PCM to 8-bit logarithmic PCM using either A or μ encoding laws. With 8-bits being used to represent each sample the data rate for speech is reduced from 96 kbit/s to 64 kbit/s when sampled at 8 kHz
ADPCM (G.726)	40, 32, 24, 16	This standard is based on the G.723 and G.721 standards It has the added flexibility of allowing the difference signal to be coded using only 2 bits, thus resulting in a data rate of 16 kbit/s (16 kbit/s overload channels carrying voice in DCME)
ADPCM (G.727)	40, 32, 24, 16	The standard provides conversion of a 64 kbit/s PCM channel to and from variable rate embedded ADPCM channels, at rates of 40, 32, 24 and 16 kbit/s. The standard has been developed to allow the difference signal to be represented by a set of core bits and enhancement bits. The core bits are the essential part for transmission. In the presence of congestion the enhancement bits may be dropped from the transmission so that the essential core bits are not lost

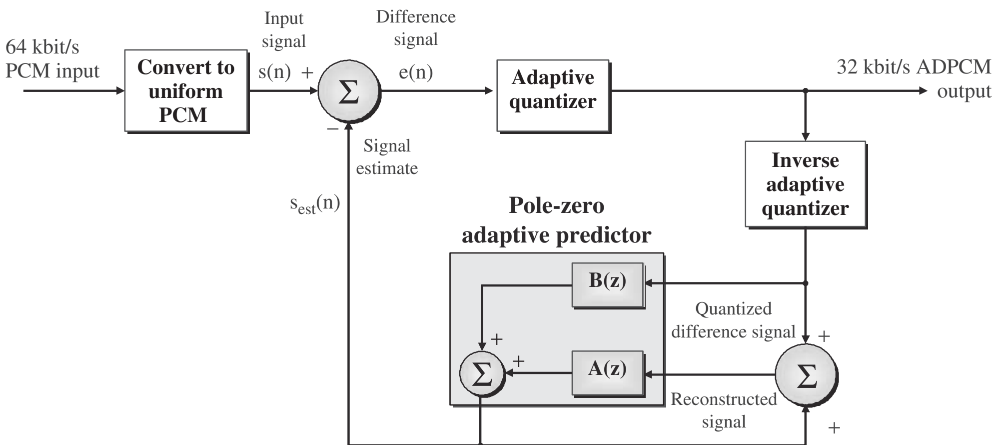


Figure 10.16 Basic structure for ADPCM coding

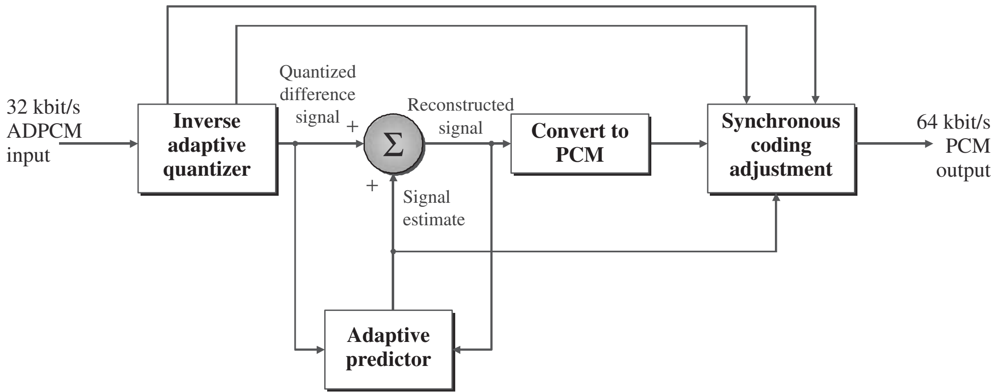


Figure 10.17 Basic structure for ADPCM decoding

giving the opportunity to develop a single device with dual functionality. Alternatively, separate modules could be developed for the encoding and decoding functions. Scalability can be achieved by scaling the memory and addressing logic, or by implementing multiple instantiations of the encoder or decoder blocks if needed.

A summary of the implementation considerations for an IP core for ADPCM is given below:

Range of Applications to be Supported

Applications range from simple DECT phones to high-end telecommunications systems such as OC-192 supporting over 1000 voice channels. Scalability would need to be incorporated to allow use of the core computation blocks for multiples of channels (Figure 10.18). Likewise, multiple instantiations of the core computational block may be needed to meet aggressive channel demands. Note that speech has set requirements on data rate and the sampling rate that can be delivered. Adding additional channels requires that the key hardware can manage the computation meeting real-time voice requirements.

Variations of ADPCM Standard

There are a number of variants of the ADPCM and PCM standards, some of which are listed in Table 10.2, and illustrated in Figure 10.19. It is wise to support an extensive variation of these standards, as this will increase the widespread suitability. For example, PCM has two variants, referred to as A-law and μ-law. The A-law variant is the standard within Europe whereas the μ-law is the standard in the US.

Supporting a Range of FPGA Devices and/or ASIC Technologies

By including additional code and parameters, the same core design can be retargeted to a different technology which can enable a design to be prototyped on FPGA before targeting to ASIC. It would also allow for low-yield implementations that would not warrant the ASIC design overhead.

Likewise, there is a great benefit in the ability to quickly redevelop the core for emerging ASIC foundries.

Ability to Support a Range of Performance Criteria

Variation in voice compression applications creates a wide span of desired features. For some applications, for example, mobile communications, power consideration and chip area could be the driving criteria for the device. For others, a high-data-rate system could be the primary objective.

Scalable Architecture

To create the flexibility needed to support such a wide range of design criteria, a scalable architecture needs to be developed that can increase the level of physical hardware to match the needs of the specification. Some key points driving the scalable architecture are:

- desired data rate
- area constraints
- clock rate constraints
- power constraints

Figure 10.18 illustrates possible architectures for scalable design.

Clock Rate Performance

The required clock rate for the system is dependent on the architecture design and the target technology. Specifying the system requirements enables the designer to make a choice regarding the target technology and facilitates a compromise with other performance criteria such as power and area.

Level of Pipelining

The desired clock rate may rely on pipelining within the design to reduce the critical path. The choice of pipelining within the submodules of the design will have a great influence over performance. For example, consider a module that performs both the encoding and decoding function, in other words, it can perform the full duplex operation. Speech is sampled at 8 kHz. For a multi-channel implementation all the computation for the encoding and decoding of all the channels needs to be carried out before the next sample, that is, within $1/8000$ s (0.000125 s). For an example module with a maximum clock rate of 300 MHz there will be 37 500 clock cycles in a 0.000125 s time frame. If in this example we consider that a duplex operation takes 20 clock cycles then the number of possible channels that can be supported is then $(300000000 \div 8000) \div 20 = 1875$ duplex channels. That is 1875 encode and 1875 decode channels.

If, for example, the module was more heavily pipelined so to reach a clock rate of 500 Hz and this increased the number of cycles per a duplex operation to 25, the number of duplex channels supported can be estimated as $(500000000 \div 8000) \div 25 = 2500$.

Figures 10.18 and 10.19 illustrate the range of applications that a parameterized ADPCM IP core would need to support.

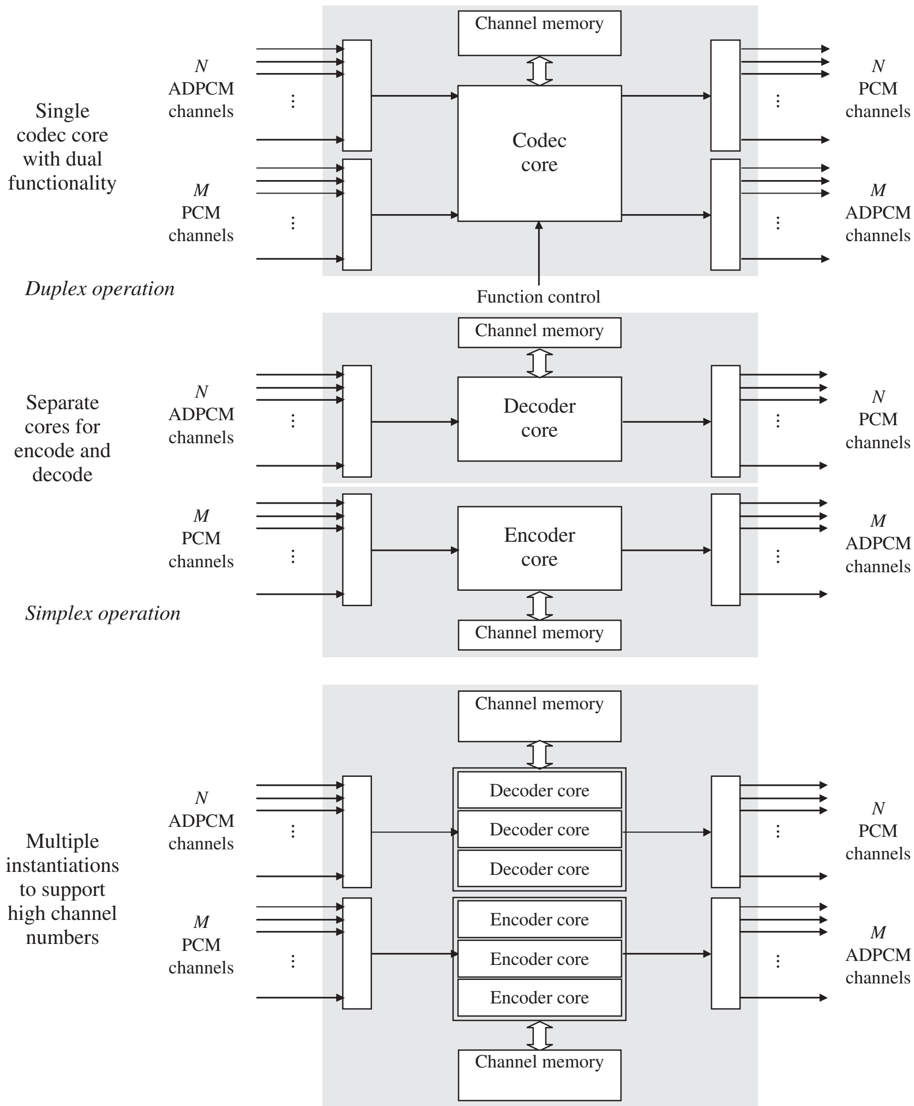


Figure 10.18 Multichannel ADPCM

10.7 Current FPGA-based IP Cores

FPGA companies identified earlier on the importance of IP cores to assist in developing solutions using their technology. For this reason, they began to develop their in-house IP libraries, but quickly identified that there were advantages to be had in developing relationships with third party vendors,

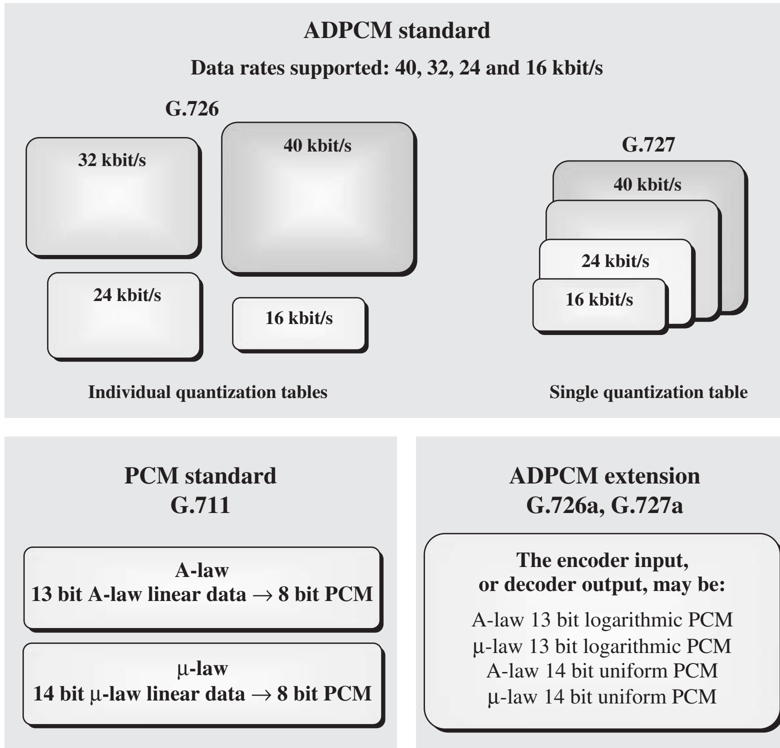


Figure 10.19 Variations of PCM and ADPCM

as both parties could benefit from the relationship, one because they would receive income for their cores and the other because the availability of the IP was making the solution easier for the customer. Both the main vendors have active IP programs, Xilinx in the form of the Alliance program and Altera in the form of the Megafunctions Partners Program; Actel have a CompanionCore Alliance Program and Lattice have the ispLeverCORE™ Connection. In all cases, companies must meet a number of criteria to become members of the respective IP programs.

Table 10.3 gives the full range of third party IP available in the areas of processors, interfaces, DSP and communications. As can be seen from the table, soft cores are available for at least one form of a microprocessor or microcontroller. The interfaces give a wide range of most common interface technologies which is vitally important, given the use of FPGAs in telecommunications applications. The DSP list comprises most of the functions listed in this book, along with complex systems such as JPEG and MPEG.

Of course, as was highlighted in this chapter, there can be changes from technology offering to technology offering and there is some work in migrating the cores to the latest technology. Moreover, some of the IP cores may no longer be actively supported by the component. Take for example, Amphion, who have been bought out by Conexant; the encryption and encryption cores are no longer their core business, but will be still be offered.

Table 10.3 Complete third party IP available from Altera

Technical area	IP available
Processors	NIOSII, ARM Cortex, 2901 processor, C68000 microprocessor, CZ80CPU, R8051 microcontroller, DF6811 CPU microcontroller
Interfaces	DDR/DDR2 RAM, 32/64-bit PCI, PCI express, Rapid I/O, DMA, CAN, UART, I2C, Gigabit Ehternet, 10/100/1000 Ethernet, ATA-4/5, AMBA, USB
DSP	FIR filter, FFT, Reed Solomon Decoder/encoder, Viterbi decoder, AES and DES encryption/decryption, SHA, Turbo encoder, motion JPEG, ADPCM, H264, JPEG, DCT/IDCT, DWT, PPL, digital modulators receivers, CCIR encoder/decoder
Communications	CRC compiler, POS-phy, UTOPIA, SONeT/SDH, Ethernet layer 2, Gigabit ethernet MUX, ATM format/deformat, AA5, HDLC, SPI-4

Table 10.4 Third party IP available for Xilinx Virtex FPGAs

Area	Function	Provider
Encryption/decryption	SHA, MD5 Hashing, AES AES Enc./Dec., TDES, RSA	Helion Tech. Limited Hidea Sol. Co., Ltd, CAST, Inc.
Video	H.264/AVC decoder, deblock H.264/AVC deblock JPEG encoder	Nero AG, Global Dig. Tech. Elecard Dev. CJSC CAST Inc., Barco-Silex
Audio processor	Sample rate conversion 32-bit APS3 processor	Coreworks CAST, Inc.
Computer	SDRAM controller Flash, SD memory controllers DDR controller Serial ATA 8-bit high speed controllers	Array Electronics Eureka Technology HCL Tech. Ltd., CAST Inc. ASICS World Ser. Ltd CAST Inc.
Telecommunications	SONET/SDH framer, digital wrapper EtherCAT slave controller ADPCM Ethernet MAC, MAC, HiGig Interlaken interconnect	Xelic, Inc. Beckhoff Auto. GmbH Pinpoint Sol. Inc. MorethanIP GmbH Sarance
Data compression	LZRW3	Helion Tech. Ltd

10.8 Summary

This chapter has provided an overview of some of the key challenges in IP design with a focus on FPGAs. The motivation behind using IP cores and design for reuse practices is covered, with discussion of the ever-widening design productivity gap. This point is highlighted in the ITRS

2005 report (Semiconductor Industry Association 2005) where it states that 'to avoid exponentially increasing design cost, overall productivity of designed functions on chip must scale at $> 2\times$ per technology generation'. To do this, design reuse is a necessity which is in effect, what IP cores capture. For this reason, we have seen a remarkable growth in the IP industry.

To some extent, many of the techniques described in Chapter 8 become the cornerstone for the development of such techniques. The major goal is to develop methodologies and flows which derive circuit architectures in such a way that aspects such as regularity and scaling are naturally captured when translating the algorithmic description into hardware. However, this is not straightforward and for this reason, Chapter 12 has been dedicated to the development of a silicon IP core for RLS filtering.

As the IP core problem has evolved from the creation of simple arithmetic cores, to the creation of system components such as FFT, DCT and DWT cores, right up to the development of complete systems such as JPEG and MPEG encoders, so therefore has the design problem. The methods in Chapter 8 are successful in creating efficient architectures for complex components such as FIR filters, DCT cores and the like, but as the work in Chapter 9 quickly demonstrated, this was insufficient for system descriptions. Therefore the design problem has changed and we need to develop higher-level techniques as the problem grows. However, any future design flow must also attempt to incorporate the IP cores as this considerable body of design work cannot be ignored. Thus, the next chapter acts to address this issue.

References

Alaraje N and DeGroat J (2005) Evolution of re-configurable architectures to SoFPGA. *48th Midwest Symp. on Circuits and Systems* pp. 818–821.

Association SI (2005) International technology roadmap for semiconductors: Design. Web publication downloadable from http://www.itrs.net/Links/2005ITRS/Design2005.pdf.

Association SI (2006) International technology roadmap for semiconductors: Design and systems drivers. Web publication downloadable from http://www.itrs.net/Links.

Birnbaum M (2001) VSIA Quality Metrics for IP and SoC. *Int. Symp. on Quality Electronic Design*, pp. 279–283.

Birnbaum M and Sachs H (1999) How VSIA answers the SOC dilemma. *Computer* **32**(6), 42–50.

Chiang S and C. T (2001) Foundries and the dawn of an open IP era. *IEEE Computer* **34**(4), 43–46.

Coussy P, Baganne A and Martin E (2002) A design methodology for IP integration. *Proc. Int. Symp. on Circuits and Systems*, pp. 127–130.

Coussy P, Casseau E, Bomel P, Baganne A and Martin E (2006) A formal method for hardware IP design and integration under I/O and timing constraints. *ACM Transactions on Embedded Comp. Syst.* **5**(1), 29–53.

Coussy P, Casseau E, Bomel P, Baganne A and Martin E (2007) Constrained algorithmic IP design for system-on-chip. *Journal of Integr. VLSI* **40**(2), 94–105.

Craven S and Athanas P (2007) Examining the viability of FPGA supercomputing. *EURASIP Journal on Embedded Systems* **1**, 13–13.

Davis L (2006) Hardware Components, Semiconductor–Digital–Programmable Logic IP Cores. Web publication downloadable from http://www.interfacebus.com/IP_Core_Vendors.html.

Ercegovac MD and Lang T (1987) On-the-fly conversion of redundant into conventional representations. *IEEE Trans. Comput.* **36**(7), 895–897.

Erdogan A, Hasan M and Arslan T (2003) Algorithmic low power FIR cores. *IEE Proc. Circuits, Devices and Systems* **150**(3), 155–160.

Gajski D, Wu AH, Chaiyakul V, Mori S, Nukiyama T and Bricaud P (2000) Essential issues for IP reuse. *Proc. ASP Design Automation Conf.*, pp. 37–42.

Guo JJ, Ju R-C and Chen J-W (2004) An efficient 2-D DCT/IDCT core design using cyclic convolution and adder-based realization. *IEEE Trans. Circuits and Systems for Video Tech.* **14**(4), 416–428.

Howes J (1998) IP New Year. *New Electronics* **31**(1), 41–42.

Huang Y, Cheng W, Tsai C, Mukherjee N, Samman O, Zaidan Y and Reddy S (2001) Resource allocation and test scheduling for concurrent test of core-based SOC design. *Proc. IEEE Asian Test Symposium (ATS)*, pp. 265–270.

Hunter J (1999) *Rapid Design of DCT cores*. PhD Dissertation, School of Electrical and Electronic Engineering, Queen's University of Belfast.

Hwang K (1979) *Computer Arithmetic: Principles, Architecture and Design*. John Wiley & Sons, Inc., New York, NY, USA.

IRTS (1999) *International Technology Roadmap for Semiconductors*, 1999 edn. Semiconductor Industry Association. Web publication downloadable from http://public.itrs.net

Junchao Z, Weiliang C and Shaojun W (2001) Parameterized IP core design. *Proc. 4th Int. Conf. on ASIC*, pp. 744–747.

Koren I (1993) *Computer arithmetic algorithms*. Prentice-Hall, Upper Saddle River, NJ, USA.

Lightbody G, Woods R and Francey J (2007) Soft IP core implementation of recursive least squares filter using only multiplicative and additive operators *Proc. Int. Conf. on Field Programmable Logic*, pp. 597–600.

McCanny J, Hu Y, Ding T, Trainor D and Ridge D (1996) Rapid design of DSP ASIC cores using hierarchical VHDL libraries. *30th Asilomar Conf. on Signals, Systems and Computers*, pp. 1344–1348.

Moretti G (2001) Your core, my design, our problem, *EDN*, 10 November 2001, pp. 57–64.

Rowen C (2002) Reducing SoC simulation and development time. *Computer* **35**(12), 29–34.

Schwarz EM and Flynn MJ (1993) Parallel high-radix nonrestoring division. *IEEE Trans. Comput.* **42**(10), 1234–1246.

Sekanina L (2003) Towards evolvable IP cores for FPGAs. *Proc. NASA/DoD Conf. on Evolvable Hardware*, pp. 145–154.

Shi C and Brodersen R (2003) An automated floating-point to fixed-point conversion methodology. *IEEE Int. Conf. on Acoustics, Speech, and Signal Proc.* **2**(2), 529–32.

Soderquist P and Leeser M (2004) Technology Roadmap for Semiconductors. *IEEE Computer* **37**(1), 47–56.

Srinivas H and Parhi K (1992) A fast VLSI adder architecture. *IEEE Journal of Solid-State Circuits* **27**(5), 761–767.

Takagi N, Yasuura H and Yajima S (1985) High-speed VLSI multiplication algorithm with a redundant binary addition tree. *IEEE Trans. Comput.* **34**(9), 789–796.

University N (2007) Variable precision floating point modules. Web publication downloadable from http://www.ece.neu.edu/groups/rpl/projects/floatingpoint/index.html.

Varma P and Bhatia S (1998) A structured test re-use methodology for core-based system chips. *Proc. IEEE Int. Conf. on Test.*, pp. 294–302.

VSIA 2007 Vsia quality IP metric. Web publication downloadable from http://www.vsia.org/documents/.

Walke R (1997) High sample rate givens rotations for recursive least squares. PhD Thesis, University of Warwick.

11

Model-based Design for Heterogeneous FPGA

11.1 Introduction

The material to this point has clearly indicated the requirement for a shift toward higher-level representations of FPGA-based DSP systems and the need for carrying out optimizations at this higher level. The material in Chapter 6 covered *lower-level* design techniques which were aimed at producing efficient FPGA implementations of circuit architectures. These design techniques covered not only the choice of how to map memory requirements into LUT and embedded RAM resources and the implications that this has on performance, but also the use of distributed arithmetic and reconfigurable multiplexers to reduce hardware costs. However, the evolution of FPGA technologies toward coarser-grained heterogeneous platforms, i.e. those comprising processors and complex DSP blocks involving dedicated multipliers and MAC units, has negated the impact of many of these latter techniques.

The material in Chapter 8 and the subsequent tool development work in Chapter 9, showed how it is possible to explore levels of parallelism and pipelining at SFG and DFG descriptions of DSP system, rather than at the lower-level HDL-based circuit architecture. It was shown how levels of parallelism and pipelining could be adjusted to best match the performance requirement of the application under consideration against the processing and memory resources of the FPGA technology available. This assumes that the SFG representation effectively represents the computational needs of a DSP system, but this is not the case. Take for example, a system implementation which requires a number of FIR filters in different places in the system; the SFG limitations requires that each filter requires separate hardware unless hardware sharing has been explicitly created in the SFG representation. It should not be an issue of how the system is described, but a system optimization that the user could investigate.

Thus, it is clear that a higher-level representation is required than SFG that allows the user to investigate the impact of system-level partitioning and system-level optimizations such as the hardware-sharing just proposed, to be explored. Such approaches form part of a larger trend toward *model of computation* (MoC)-based design. *MoCs* are used to express characteristics of systems such as timeliness, i.e. how the system deals with the concept of time, concurrency, liveness, heterogeneity, interfacing and reactivity (Lee and Sangiovanni-Vincentelli 1998). They attempt to define how a system works and how it interacts with the real, analogue world. A plethora of models has been suggested for rigorous modelling of different types of embedded systems.

FPGA-based Implementation of Signal Processing Systems R. Woods, J. McAllister, G. Lightbody and Y. Yi
© 2008 John Wiley & Sons, Ltd

Determining the appropriate MoC for modelling of certain types of system should be based on the specific characteristics of that system. For instance, a general characterization of DSP systems could describe systems of repetitive intensive computation on streams of input data. As such, DSP system developers frequently come to the same conclusion on this choice and use the dataflow MoC (Najjar *et al*. 1999). The most widely used dataflow domains, and the manner in which they enable system-level design and optimization of embedded DSP systems, are outlined in this section.

The chapter gives a detailed analysis of how dataflow MoC can be used to create FPGA-based DSP systems and illustrated this with some innovative work in creating the features of such a system and demonstrating it using suitable examples. Section 11.2 which gives an overview of dataflow modelling and highlights some of the different flavours; in particular, it covers the challenges of rapidly implementing FPGA-based DSP systems. Section 11.3 describes how changes in DFG descriptions can have a major impact in embedded system implementation. In Section 11.4, the reader is given a detailed treatment of the synthesis of system descriptions using pipelined cores, without the need for the redesign of these cores. The design of the necessary control and wrappers for these cores is then outlined in Section 11.5. Two examples, namely a normalized lattice filter and a fixed beamformer, are then used to demonstrate the approach in Section 11.6. Finally, concluding remarks are made in Section 11.7.

11.2 Dataflow Modelling and Rapid Implementation for FPGA DSP Systems

The roots of the most popular current dataflow languages lie in the Kahn process network (KPN) model (Kahn 1974). The KPN model describes a set of parallel processes (or ‘computing stations’) communicating via unidirectional first-in first-out (FIFO) queues. A computing station consumes data tokens coming along its input lines, using localized memory, producing output on one or all of its output lines. In DSP systems, the tokens are usually digitized input data values. Continuous input to the system generates streams of input data, prompting the computing stations to produce streams of data on the system outputs. The general structure of a KPN is shown in Figure 11.1. The semantics of repetitive application of specific computing functions to every input sample in KPN makes this modelling approach a good match with the behaviour of DSP systems.

In the dataflow process network (DPN) model (Lee and Parks 1995), the KPN model is augmented with semantics for computing station (known here as actor) behaviour. A sequence of actor firings is defined to be a particular type of Kahn process called a dataflow process, where each firing maps input tokens to output tokens, and a succession map input streams to output streams. A set of firing rules determine, for each actor, how and when it fires. Specifically, actor firing *consumes*

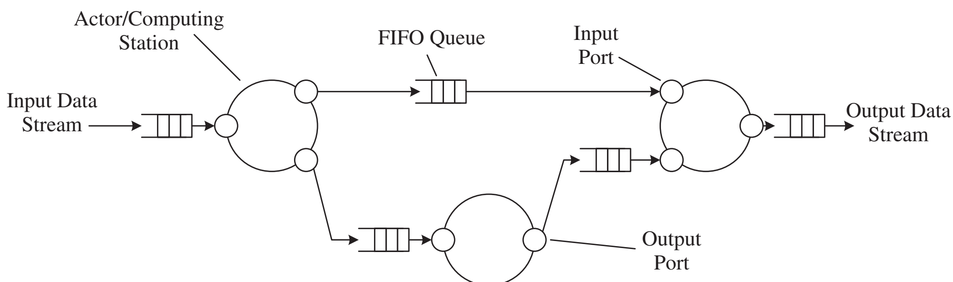


Figure 11.1 Simple KPN Structure

input tokens and *produces* output tokens. A set of sequential firing rules exist for each actor, and define the input data conditions under which the actor may fire. Given the solid computational foundation of DPN, which describes how actors fire and communicate deterministically, numerous application-specific refinements on this general theme have been proposed. Three specific variants are of particular importance in this section, synchronous dataflow (SDF), cyclo-static dataflow (CSDF) and multidimensional synchronous dataflow (MSDF). The semantics of these three are outlined now.

11.2.1 Synchronous Dataflow

Lee defines a synchronous dataflow (SDF) system (Lee and Messerschmitt 1987b) to be one where ‘we can specify *a priori* the number of input samples consumed on each input and the number of output samples produced on each output each time the block is invoked’. This is not the definition of synchronicity supplied for general MoC definition in (Lee and Sangiovanni-Vincentelli 1998); nevertheless, this knowledge allows derivation of a static system schedule (i.e. one that is generated at compile time), which is important because it means that multiprocessor schedules with low run-time overhead can be created. This was a significant advance in the dataflow MoC, and pioneered a large body of research into dataflow system modelling and implementation techniques. However, this advantage is gained at the expense of expressive power since SDF forbids data-dependent dataflow behaviour.

Each port j on an actor i in an SDF graph has an associated *firing threshold* tf_{i_j} which specifies the number of tokens consumed/produced (depending on the direction of the port) at that port in a firing of the actor. This value is quoted adjacent to the port, as illustrated in Figure 11.2. When all port thresholds in an SDF are unity, the graph is known as *homogeneous*, or a single-rate DFG (SRDFG). Otherwise, the DFG is known as a multi-rate dataflow graph (MRDFG).

If, for actor j connected to arc i, $x_j^i(n)$ is the number of tokens produced and $y_j^i(n)$ is the number consumed at the nth invocation of the actor, an SDF graph can characterized by a topology matrix Γ (Equation 11.1).

$$\Gamma_{ij} = \begin{cases} x_j^i(n) & \text{if task } j \text{ produces on arc } i \\ -y_j^i(n) & \text{if task } j \text{ consumes from arc } i \\ 0 & \text{otherwise} \end{cases} \tag{11.1}$$

A key element of proving the consistency of a DFG is the satisfaction of a set of balance equations (Lee 1991). Actor A fires proportionally $q_{(A)}$ times in an iteration of the schedule and produces t_A tokens per firing, and actor B fires proportionally $q_{(B)}$ times and consumes t_B tokens per firing. If A is connected to B, in an iteration of the schedule all FIFO queues return to their initial state (Lee and Messerschmitt 1987a), then Equation (11.2) will hold. Collecting such an equation for each arc in the graph, a system of balance equations is constructed, which is written compactly as Equation (11.3). In Equation (11.3), the *repetitions vector* q describes the number of

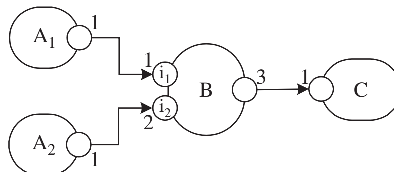


Figure 11.2 Simple SDF graph

firings of each actor in an iteration of the execution schedule of the graph, where entry q_i is the number of firings of actor i in the schedule.

$$q(A)t_A = q(B)t_B \tag{11.2}$$

$$\Gamma q = 0 \tag{11.3}$$

11.2.2 Cyclo-static Dataflow

The cyclo-static dataflow (CSDF) model (Bilsen *et al.* 1996) notes the limitation of the MRDFG that only invariant actor behaviour from firing to firing can be supported. It expands the SDF model to allow a cyclically changing actor behaviour, whilst still maintaining the static schedulability capabilities. The firing activity of each actor is generalized where every actor v_j now has a sequence of j firings $\gamma = [f_j(1), f_j(2), \ldots, f_j(P_j)]$ each of which has different firing rules, with γ_i occurring on the i (mod j)$^{\text{th}}$ firing of the actor, creating the scenario where the sequence of firings is repeatedly cycled through in turn. The scalar production and consumption values of actor ports in the SDF graphs are replaced with vectors of length P, where P_i defines the number of tokens consumed/produced at that port on the ith firing of the actor. If, for actor j connected to arc i, $X_j^i(n)$ is the total number of tokens produced and $Y_j^i(n)$ the total number consumed during the first n invocations of the actor, a CSDF topology matrix Γ is defined as in Equation (11.4). Figure 11.3 shows an example CSDF graph with the threshold vector enclosed in rectangular braces adjacent to the port.

$$\Gamma_{ij} = \begin{cases} X_j^i(P_j) & \text{if task } j \text{ produces on arc } i \\ -Y_j^i(P_j) & \text{if task } j \text{ consumes from arc } i \\ 0 & \text{otherwise} \end{cases} \tag{11.4}$$

11.2.3 Multidimensional Synchronous Dataflow

The atomic token principle of SDF mean multidimensional systems (i.e. those featuring computations on multidimensional tokens such as vectors or matrices) must be collapsed onto graphs using unidimensional streams, a restriction which can hide data-level parallelism in the algorithm. To overcome this limitation, the SDF model was generalized to multidimensional synchronous dataflow (MSDF). Initial work (Lee 1993a,b) evolved the SDF domain to sampling on rectangular lattice shaped problems, but later (Murthy and Lee 2002) extended the technique to arbitrarily shaped lattices. Token production and consumption are now specified as M-tuples, and the number of balance equations per arc increased from 1 to M. An example MSDF graph and its associated balance equations are given in Figure 11.4 (Lee 1993a) where the M dimensions of the tokens produced and consumed, are given in circular braces adjacent to the port. This domain provides an

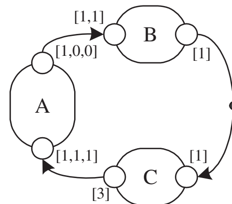


Figure 11.3 Simple CSDF graph

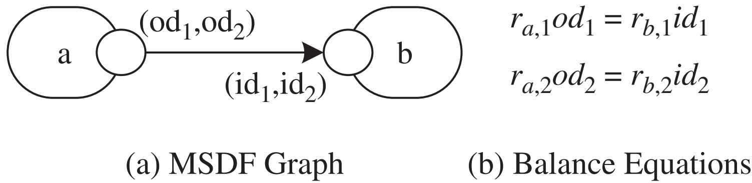


Figure 11.4 MSDF graph and association balance equations

elegant solution to multidimensional scheduling problems in SDF graphs, and exposes additional intra-token parallelism for higher-order dimension tokens (Lee 1993a).

11.2.4 Dataflow Heterogeneous System Prototyping

Typically, heterogeneous system design for system-on-chip (SOC) and FPGA requires an incremental refinement from a high-level system model to implementation (Keutzer *et al*. 2000). This requires resolution of issues such as communication refinement (Rowson and Sangiovanni-Vincentelli 1997) and dedicated hardware synthesis, amongst a plethora of others. Alternatively by closely matching the behavioural semantics of the implementation to the semantics of the high-level specification, a direct translation from the functional to the implementation domains can be enabled. This is a rapid system integration technique. A general overview of such a desired methodology is outlined in Figure 11.5, and this shows the three key requirements for such an approach:

1. A suitable modelling domain for the specification
2. A rapid integration approach
3. Capability for analysis and high-level optimization of the implementation

The high-level algorithm model should ideally employ infinitely precise mathematics, but the architecture of modern workstation technology where this modeling is performed in practice means that integer and single- or double-precision floating-point mathematics are used in modelling applications such as MATLAB, which is usually sufficient for most applications. After ideal algorithm verification the numerical modelling is then adapted to incorporate reduced precision numerical modelling (fixed/floating-point) and wordlength minimization with a view to identifying candidate functions for implementation as dedicated hardware components. The importance of this type of operation is acknowledged by the incorporation of reduced precision mathematical modelling in system design languages such as SystemC (Grötker *et al*. 2002), and tools such as MATLAB.

The system functionality is then mapped to one of the physical processors on the embedded platform. This partitioning is usually based on physically constraining implementation factors such as throughput requirements, available dedicated hardware resources, communications bandwidth or power consumption (Gajski *et al*. 1994, Kalavade and Lee 1997). After partitioning, the implementation begins.

11.2.5 Partitioned Algorithm Implementation

Implementation requires rapid translation of the partitioned algorithm to networks of interacting hardware and software portions. The general structure of this integration approach is shown in Figure 11.6. The partitioned algorithm specifies pools of functionality to be implemented on one or more microprocessors, pools to be implemented on one or more FPGAs, and inter-processor communication points. This describes the entire embedded system at a functional level. This description

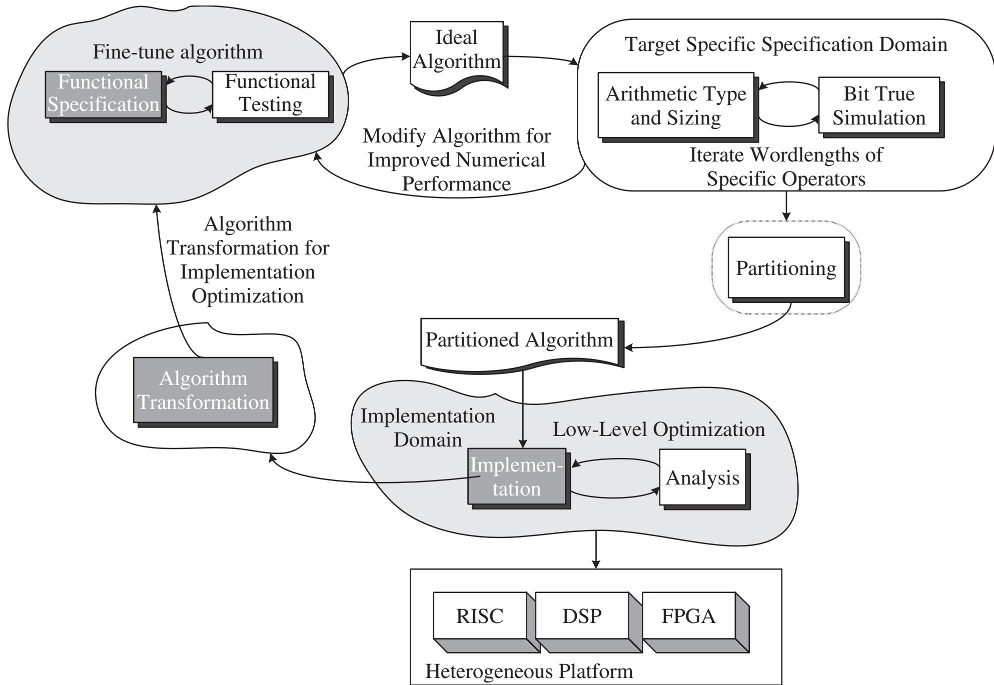


Figure 11.5 DSP system design methodology

then requires translation to a physical manifestation. Actors are collected into pools, before resolution of three major issues:

1. Software synthesis for each of the embedded microprocessors
2. Hardware synthesis for each FPGA
3. Inter-processor communication fabric insertion

For design time minimization, the designer can make use of the extensive and growing libraries of precomposed functionality for efficient implementation of specific actors in both hardware and software. In particular for FPGA designers, there now exist extensive libraries of intellectual property (IP) components (cores), parameterized in terms of features such as word size to encourage reuse in multiple system instances. This methodology emphasizes core-based design due to the potential for reduced design time offered by use of these components, since implementation effort reduces to core integration.

There are however, a number of issues with such an approach. Integrating multiple different cores from multiple different sources into a single FPGA core network means that interfacing issues may occur. Despite efforts by standards bodies such as the Virtual Socket Interface Alliance or VSIA (VSIA 2007), no standard interface protocol for cores has been widely adopted, meaning that the designer must impose one for a particular application. Additionally, in the MoC-based design approaches used in this methodology, there may be behavioural flexibility demands made of the core, not addressed by the core synthesis procedure. These are considered in more detail later

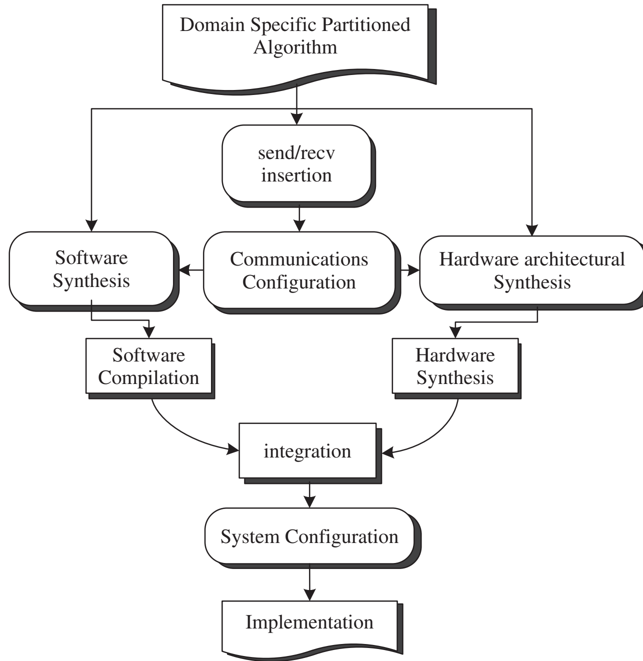


Figure 11.6 DSP system rapid implementation

in this chapter. In the remainder of this section the processes of inter-processor communication refinement and software synthesis are described.

11.3 Rapid Synthesis and Optimization of Embedded Software from DFGs

A DFG $G = V, E$, describes a set of vertices (actors) V interconnected by a set of edges E. Recalling that the edges are conceptually FIFO token queues, where the tokens can be any arbitrary numerical storage unit, then in translation from algorithm to a software routine executing on an embedded microprocessor there are four main stages, as outlined in Figure 11.7. Consider the four steps in the context of the example DFG in Figure 11.8.

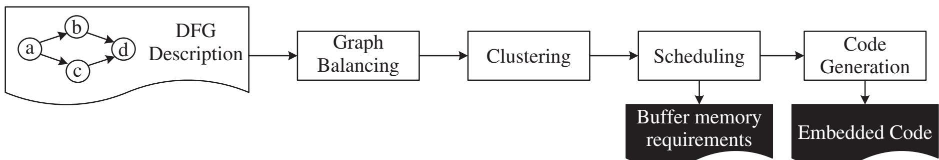


Figure 11.7 Rapid software synthesis from DFGs

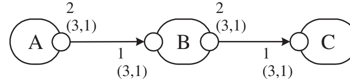


Figure 11.8 Example DFG

11.3.1 Graph-level Optimization

The first level of optimization occurs by applying *algorithmic engineering* techniques to the DFG for implementation optimization. This is particularly applicable to system modelling using MSDF semantics (Murthy and Lee 2002). Consider the simple equivalent DFGs in Figure 11.9. Assuming token atomicity, then the maximum possible partitioning of the DFG of Figure 11.9(a) is to a single processor, with all the communication channels implementing vector transfers. This hides some of the data parallelism inherent in the vector multiplication algorithm. For an n-element vector, processing the vector multiplication may require n cycles, a critical drawback for high-throughput implementation. Alternatively, the token may be specified as a scalar as in Figure 11.9(b), allowing an n-fold increase in multiprocessor partitioning and corresponding throughput. This manipulation also alters the required memory resource for each processor, since each now requires buffering for scalar tokens rather than n-element vectors. The drawbacks include increased scheduling complexity when the intra-token parallelism is exploited. This optimization may be particularly applicable to FPGA hardware component network synthesis, since the structure of the computational resource on FPGA is under control of the designer. Hence, this may offer a very promising architectural exploration approach for DFG actor networks mapped to FPGA.

Notice that at this level, to enable the kinds of optimizing transformation described here, the graph must be flexible, primarily in terms of the tokens it processes: i.e. the token dimensions x_i processed at port i.

11.3.2 Graph Balancing Operation and Optimization

In the domain of dataflow-based rapid embedded system design, undoubtedly the two most significant advances were the establishment of the SDF domain, and associated potential for minimal

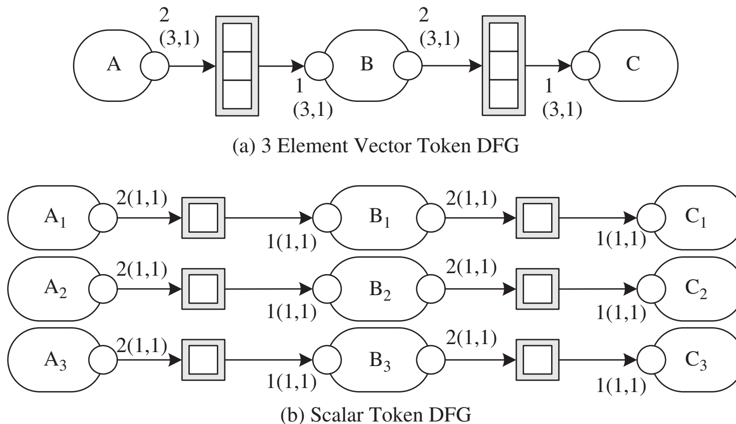


Figure 11.9 Scalar/vector DFG comparison

run-time overhead embedded execution, and the introduction of formal analysis methods for consistency of DFGs in terms of ensuring memory-bounded and deadlock-free execution (Lee 1991). All the other, more expressive dataflow domains in this section stem from generalization of these ideas, and in particular Equations (11.2) and (11.3). For instance, CSDF merely generalizes the scalar q and r in Equation (11.2) to two-dimensional objects, and exploits the space afforded by the extra dimension to affect variable actor firing activity, exploiting previously hidden parallelism and enabling more effective use of memory resources, without violating the consistency conditions in Equations (11.2) and (11.3) (Parks *et al.* 1995).

During the first stage, of translation from the application DFG to an embedded implementation, *graph balancing* is performed, i.e. the consistency conditions in Equations (11.2) and (11.3) (Parks *et al.* 1995) are satisfied. For the example SDF graph of Figure 11.8, with actor and arc numbering as in Figure 11.10, Γ and **q** are given in Equations (11.5) and (11.6), respectively.

$$\Gamma = \begin{bmatrix} 2 & -1 & 0 \\ 0 & 2 & -1 \end{bmatrix} \tag{11.5}$$

$$q^{\mathrm{T}} = \begin{bmatrix} 1 & 2 & 4 \end{bmatrix} \tag{11.6}$$

At the balancing level of abstraction, the main optimization opportunities arise as a result of implementing *block processing* (Lee and Messerschmitt 1987a). Essentially, this involves application of an integer scaling factor to **q** to scale the number of firings of each actor by an integer number. Once this is applied, lower-level transformations can be applied to increase implementation efficiency and performance, as outlined in Section 11.3.4.

11.3.3 Clustering Operation and Optimization

Numerous valid schedules exist for the example DFG of Figure 11.8, two valid examples being $S_1 =$ *ABCCBCC* and $S_2 =$ *ABBCCCC*. Note that these two schedules have different ordered groupings of the various actor firings. Within a given balanced manifestation of a dataflow algorithm, q_i defines the number of firings of actor i in an iteration of the schedule. At the clustering stage, the set of firings of each actor is subdivided into groups of one or more consecutive firings, known as *executions*. The number of firings per execution of actor i is known as its *granularity* and defines the number of tokens which must be available at the input to each port of the actor prior to execution. For the example schedules S_1 and S_2, the dependence graphs, outlining the firings and executions of each actor are given in Figure 11.11.

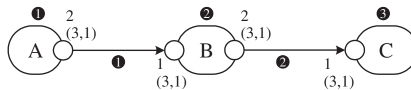


Figure 11.10 Numbered example DFG

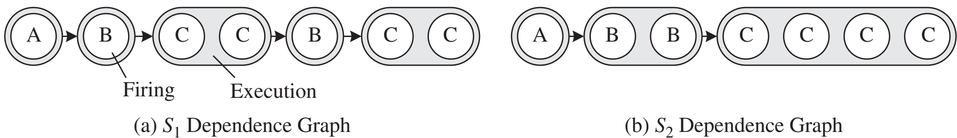


Figure 11.11 Schedules firing/execution organization

Reinterpreting Equation (11.2) in terms of execution thresholds and invocations allows the definition of a balanced system as Equation (11.9), where Γ_e is given as in equation (11.7) and q_e, the execution repetition vector, is derived by solving Equation (11.8) for each DFG arc. Note particularly the inclusion of a scaling factor k_j associated with each actor j. This is used for optimization of the clustering stage later in this section.

$$\Gamma_{ij} = \begin{cases} k_j \times x_j^i(n) & \text{if task } j \text{ produces on arc } i \\ -k_j \times y_j^i(n) & \text{if task } j \text{ consumes from arc } i \\ 0 & \text{otherwise} \end{cases} \tag{11.7}$$

$$q_e(j) = q(j)/k_j \tag{11.8}$$

$$\Gamma_e q_e = 0 \tag{11.9}$$

Provided that Equation (11.9) is satisfied, the schedule is sample rate consistent, one of the two key features of a valid schedule (Lee 1991, see Section 11.3.4 for more details). Hence controlling the clustering stage via manipulation of Γ_e and q_e provides two degrees of freedom for implementation optimization. This manipulation is affected by manipulating the k factor associated with each actor during the clustering stage. Consider how this may enable implementation optimization in the context of the example DFG of Figure 11.8. The values of Γ_e and q_e given the actor/arc labelling in Figure 11.10 are given in Table 11.1.

The *high-g* transformation is used to maximize individual actor execution thresholds by increasing the values of k_j in Equations (11.8) and (11.7). This reduces the number of executions of each actor, and is used to cluster the actors in a manner which may reduce run-time overheads in the implementation. These overheads may arise from a variety of sources, such as IPC (where time penalties can be incurred for setting up/shutting down a communication link with a remote processor) or dynamic scheduling. Consider the first schedule, *A(2B(2C))*, in Figure 11.11(a), where the actor C is an IPC actor, sending data to a remote processor. In this scenario, the overhead associated with setting up and shutting down the transfer is invoked at the invocation and conclusion of the actor execution, respectively. To maximize the efficiency of the communication (i.e. amount of data transferred per overhead penalty invoked), the number of firings of C per execution (i.e. the granularity of C) should be maximized by increasing the value of k_c. This is affected by setting $k_c = 4$ during the clustering step, such that a single execution of C occurs, with all the firings occurring during this execution. This may result in a schedule such as *ABBCCCC*.

Consider, on the other hand, the two schedules from Figure 11.15 in terms of data memory requirements for execution initiation. In Figure 11.11(a), the high granularity of C means that a data buffer must be maintained to store all four of the tokens to be input to C before the first firing starts. In cases like this, to reduce the amount of buffer memory required for execution, *high-q* transformation may be used. In the case of Figure 11.15, this involves minimizing k_c and k_b to,

Table 11.1 Example Schedules Γ_e and q_e

Schedule	Γ	q_eT
ABCCBCC	$\begin{bmatrix} 2 & -1 & 0 \\ 0 & 2 & -2 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 2 \end{bmatrix}$
ABBCCCC	$\begin{bmatrix} 2 & -2 & 0 \\ 0 & 4 & -4 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$

for instance, $k_c = 2$ and $k_b = 1$, respectively. This reduces the number of firings in an execution, reducing the number of tokens which must be stored before execution of b and c can begin.

11.3.4 Scheduling Operation and Optimization

The final step before generating the source code for the embedded implementation of the DFG is scheduling of the executions. During this step the various executions of each actor are sequenced for sequential execution. As outlined in (Lee 1991), two key steps must be satisfied before it can be ensured that a periodic admissible schedule is achievable for the embedded schedule implementation of the DFG. The first step, verification of graph sample rate consistency is ensured during graph balancing, as outlined Section 11.3.2. The second step is to ensure that the phenomenon of *deadlock* is avoided.

A significant body of research effort has been invested in studying various scheduling techniques for optimization of the embedded implementation in terms of varying aspects, such as data/code memory (Bhattacharyya *et al*. 1999), IPC or inter-processor synchronization overhead (Sriram and Bhattacharyya 2000). For instance, the *high-q* transformation techniques outlined in Section 11.3.3 are somewhat redundant without the ability to perform an interleaved scheduling optimization on the system. In the *high-q* schedule of Figure 11.11(a), fragmenting the executions of B and C has no effect on the data memory requirements unless the scheduling step can recognize the potential to interleave the executions of B and C such that the data buffer used to store the tokens for input to C can be reused for each execution, thus halving the input data buffer memory requirements for C. Without this interleaving, *high-q* optimization has no effect on data buffer requirements of embedded software.

11.3.5 Code Generation Operation and Optimization

Given q for the example DFG in Figure 11.8, some valid schedules for this graph include: $S_1 = A(2B(2C))$; $S_2 = ABCBCCC$; $S_3 = A(2B)(4C)$ and $S_4 = A(2BC)(2C)$. The terms in parentheses in S_1, S_3 and S_4 indicate looped invocations of these schedules. Hence S_1 has one firing of A, followed by two repetitions of BCC. Likewise S_3 implements $ABBCCCC$, and S_4 implements $ABCBCCC$. The code generator inserts code portions representing the functionality of an actor at each schedule actor instance. For example, in S_2 the code generator must instantiate seven code segments, one for each actor instance in the schedule. However, in S_1, S_3 and S_4 the terms in parentheses are replaced by single instantiations in code loops, reducing the number of schedule actor instances. The code segment in Figure 11.12 is typical of that which may be generated from S_4. The different scheduling methods generate schedules requiring differing code memory space. Likewise, the buffer memories for each schedule also vary. Table 11.2 summarizes the code and buffer memory requirements for each schedule. It is clear that how the system is scheduled has a significant impact on the physical embedded manifestation.

```
code block for A
for (i=0;i<2;i++){
        code block for B
        code block for C
        }
for(i=0;i<2;i++){
        code block for C
        }
```

Figure 11.12 Looped schedule source generated from schedule S4

Table 11.2 Scheduling variants memory requirements

Schedule	Code memory size (blocks)	Buffer memory size (tokens)
A(2B(2C))	3	4
ABCBCCC	7	5
A(2B)(4C)	3	6
A(2BC)(2C)	4	5

11.3.6 DFG Actor Configurability for System-level Design Space Exploration

Post integration, the real-time and physical constraints of the specification, such as throughput, processing resource, memory usage or some other constraining factors, must be met. Should these constraints be met, the implementation process is then complete. Otherwise, a method to transform the implementation must be applied. These transformations can be applied at low abstraction levels, where the high design detail means that applying numerous different transformations can be difficult and time consuming, somewhat negating the fast design time capabilities of a rapid implementation methodology. An alternative approach is to apply high-level transformations to the algorithm to influence the embedded implementation. This is a coarse-grained exploration, but is significantly faster and less detailed than low-level transformation, and may be complemented by lower-level transformations to allow the necessary fine-grained control of the implementation. This fast design space exploration is a major benefit of current MoC-based rapid implementation approaches.

This section has shown how flexibility of dataflow actors in certain respects enables generation of different multiprocessor schedules, to influence the embedded implementation. This allows the designer to manipulate the DFG context in which an actor operates to allow system-level exploration of the embedded implementation at the graph, balancing and clustering levels. A summary of the various types of flexibility required for optimization at the various abstraction levels is given in Table 11.3.

The graph-level transformations outlined in Section 11.3.1 depend on the ability to trade off the number of actors with the token dimensions of the incoming and outgoing arcs, as defined by the symbols **A** and **X** respectively in Table 11.3. At the balancing stage, the number of firings of each actor in the schedule can be manipulated by implementing block processing, i.e. scaling the q vector in Equation (11.3) by a constant scaling factor J. Finally, at the clustering stage the actor execution scaling factor is introduced to enable *high-g/high-q* transformation, as outlined in Section 11.3.3.

11.3.7 Rapid Synthesis and Optimization of Dedicated Hardware from DFGs

Chapters 6 and 8 have described how the register-rich programmable logic present on an FPGA makes them ideal for hosting high-throughput, pipelined circuit architectures for DSP functions. Furthermore, the substantial body of research into automated design techniques for translating signal

Table 11.3 DFG actor configuration for embedded optimization

Abstraction level	Manipulation aspect	Description
	A	Number of actors
Algorithm	**X**	Connecting arcs token dimensions
	S	Number of arcs
Balancing	J	Blocking factor
Clustering	k	Actor execution scaling factor

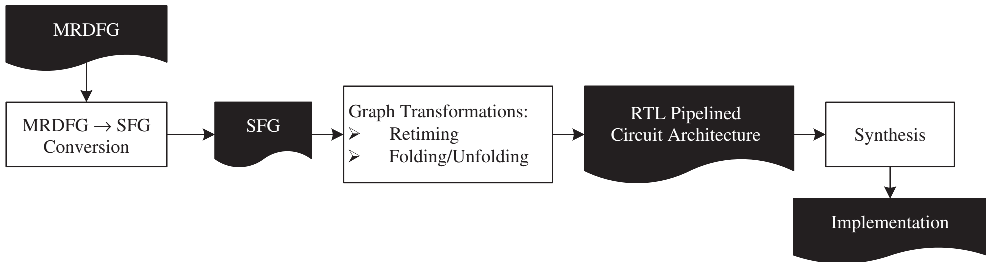


Figure 11.13 MRDFG pipelined core synthesis process

flow graphs (SFGs – a type of SRDFG) to pipelined hardware architectures (Parhi 1999), means that these design approaches can achieve high real-time performance and fit into a dataflow-centric system level design approach. These types of approach are in line with the requirements for a rapid core generation technique for this rapid implementation methodology. A typical architectural synthesis technique for translating MRDFG specifications to pipelined circuit implementations, is outlined in Figure 11.13.

The most important factor of note in this process is the first, i.e. conversion of the MRDFG description of the algorithm to a single rate SFG description. This conversion is required because the behavioural semantics of the multi-rate model are significantly more expressive than the single rate (Lee and Messerschmitt 1987b) and these semantics must be refined to enable the lower-level design processes for pipelined hardware (Parhi 1999). Specifically, there are a number of restrictions associated with SFG modeling of the system:

1. The port thresholds of all ports in the DFG are fixed at unity
2. The each actor fires only once in an iteration of the schedule
3. Port tokens are atomic

The restricted behavioural semantics of SFG as compared with MRDFG result in the more complicated semantics of the MRDFG model, and corresponding manipulations of these aspects, manifest as structural changes in the SFG graph. Architectural synthesis itself consists of three main SFG structural transformations to produce a suitable pipelined component implementation (Parhi 1999). *Retiming* processes were described earlier in Chapter 8 as well as *folding* and *unfolding*, used to optimize the circuit architecture for area or throughput constraints.

Technology for SFG architectural synthesis now includes techniques for hierarchical synthesis, advanced scheduling algorithms and incorporation of predefined cores for the lowest level (known here as primitive level) components (Yi and Woods 2006). However, the main bulk of recent work in this area has been performed in the translation from SFG to pipelined hardware architecture, largely considering the transformation from MRDFG to SFG to be a necessary precursor step. Given that the entire architectural synthesis process in Figure 11.13 is based on the SFG domain, the resulting pipelined core architecture is specific for that SFG structure, and hence by logical extension the MRDFG behavioural configuration from which the SFG was created. The restrictions imposed by this scenario are outlined in Section 11.3.8.

11.3.8 Restricted Behavioural Synthesis of Pipelined Dedicated Hardware Architectures

The problem with applying SFG synthesis approaches to actors is that alteration of any of the configuration factors in Table 11.3 does not affect the structure of the MRDFG, but conceals the

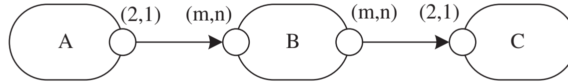


Figure 11.14 Example architectural synthesis MRDFG

changes in the behavioural semantics of the language. However, in conversion from MRDFG to SFG, these changes are then manifest as structural changes in the SFG, since the behavioural semantics are fixed here. Since the pipelined core is the result of applying architectural synthesis techniques to the SFG, this means that exploring the system design space using MRDFG actor configuration alterations requires re-synthesis of the resulting pipelined core for every explored point in the design space mapped out by the factors in Table 11.3. To illustrate the issues that arise from this restriction, consider Figure 11.14, which represents the MRDFG for a hypothetical architectural synthesis process.

Here, actor A produces tokens with dimensions (2, 1) onto the arc. These tokens are consumed by B, which is to be implemented as a pipelined hardware component. B consumes and produces tokens with dimensions (m, n), with the values of m and n under the control of the designer for algorithm level optimization as described in Section 11.3.1. The multidimensional semantics of MSDF enables token dimension exploration via manipulation of the language semantics, as outlined in Section 11.2.3. When converted to SFG however, this type of graph-level exploration of intra-token parallelism is forbidden, and manifests as structural changes in the DFG. To demonstrate this, the SFG for different values of (m, n) for B in Figure 11.14 are shown in Figure 11.15. It is assumed that the token dimensions transferred at an SFG actor port is (1, 1) with a value of G of 1, to ensure that SFG structural changes are as a result of token dimension alteration only.

In the three different configurations, altering the DFG token dimensions has altered the number of input ports on B since each SFG actor must transfer (m, n) of dimension (1, 1) in one firing. Furthermore, it is clear that varying the token dimensions has had the effect of altering q_i, the number of firings of actor i in an iteration of the graph schedule, resulting in variable numbers of actor instances since, in an SFG, $q_{(i)}$ is always 1. Hence, applying the types of graph-level transformation outlined in Section 11.3.1 without having to re-synthesize the pipelined hardware for every point in the exploration space, actors must be flexible in terms of their number of ports and the number of firing instances. This observation is reinforced by analysis of the variation of SFG structure with block processing exploration at the graph balancing stage as outlined in Section 11.3.2. Figure 11.16 shows the SFG resulting from the MRDFG of Figure 11.14, for $j = 1$

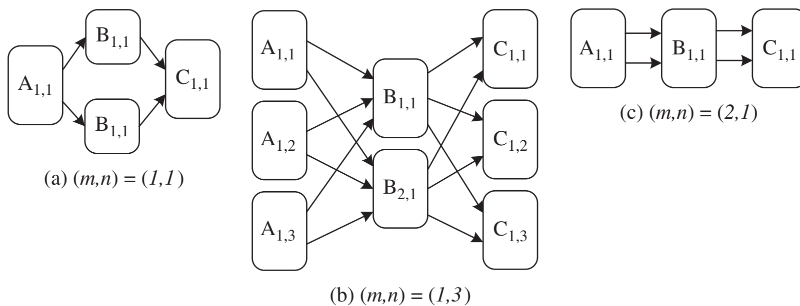


Figure 11.15 SFG structure variation with graph-level manipulation

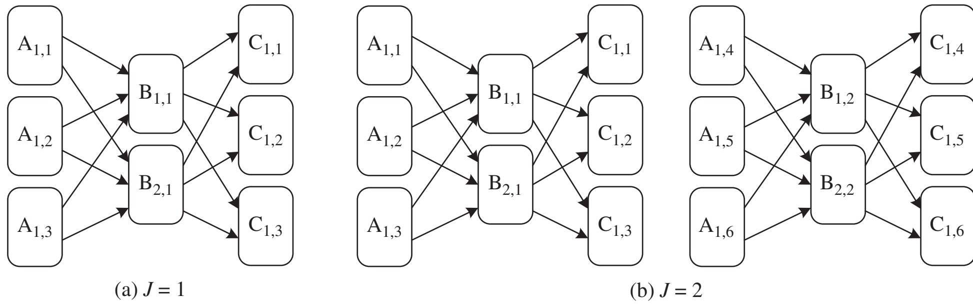


Figure 11.16 SFG structure variation with block processing factor

and $J = 2$ and constant token dimensions. As this illustrates, applying block processing effectively replicates the SFG.

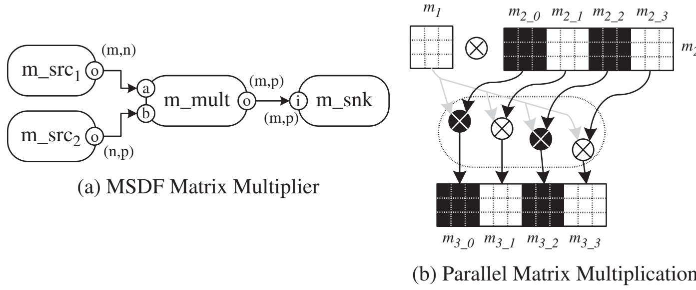
This section has shown that, due to the discrepancies between the MRDFG and the SFG behavioural semantics, SFG cannot be used for multiple MRDFG actor configurations. It logically follows that any pipelined core generated from an SFG via architectural synthesis cannot be used for different MRDFG actor configuration realizations, i.e. is not reusable in multiple MRDFG systems; every configuration change of an MRDFG actor in high-level embedded system exploration requires redesign and re-synthesis of a new pipelined core.

This is not just a drawback of SFG architectural synthesis techniques, but has ramifications for all pipelined cores when these are to be included in embedded systems, generated from MRDFGs. Hence, the dual requirements of rapid pipelined hardware component synthesis and core-based reuse of the generated components are not concurrently possible for MRDFG frameworks, given the current technology; when designing algorithms at the MRDFG level, the designer has only inferred, rather than exerted direct control on, the structure of the dedicated hardware architecture. The next section outlines a pipelined hardware core synthesis approach which overcomes these difficulties to enable generation of pipelined dedicated hardware flexible enough to implement multiple configurations of MRDFG actor without redesign, enabling true system-level exploration of heterogeneous FPGA DSP systems.

11.4 System-level Modelling for Heterogeneous Embedded DSP Systems

The key issue with lack of explicit designer control on the structure of the implementation is the lack of structural flexibility in the MRDFG itself. A single actor in standard dataflow languages such as SDF or MSDF can represent any number of tasks in the implementation, rather than employing a close relationship between the number of DFG actors and number of cores in the solution. To overcome this structural inflexibility, the multidimensional arrayed dataflow (MADF) domain may be used (McAllister *et al*. 2006). To demonstrate the semantics of the domain, consider a matrix multiplication problem – multiplication of m_1 by m_2 (dimensions (m, n) and (n, p) respectively). The MSDF graph of this problem is given in Figure 11.17(a).

By interpreting m_2 as a series of parallel column vectors, each of which is a column of m_2, the vectors can be grouped into matrices of any size, allowing concurrent multiplication of m_1 by an array of **y** matrices $m_{2_0}, m_{2_1}, \ldots, m_{2_{y-1}}$ where m_{2_i} is composed of the p column vectors $i \times p/y, \ldots, ((i+1) \times p/y) - 1$ of m_2. The subdivision of m_2 into parallel sub-matrices for $p = 4$ is given in Figure 11.17(b). Note the regular relationship between the number of multipliers and



(a) MSDF Matrix Multiplier

(b) Parallel Matrix Multiplication

Figure 11.17 Matrix multiplication MSDF and parallelism exploration

the size of sub-matrix consumed by each. This kind of relationship could be exploited to regularly change the structure of the DFG, trading off the number of actors and the token dimensions processed at the ports of each, enabling trade-off of the number of actors and the token dimensions processed by each (i.e. the same kind of transformation as enabled in MSDF) whilst still controlling the algorithm 'structure'. Given an appropriate synthesis methodology, this could be extended to influence the number of FPGA cores and token dimensions processed by each, and used to trade resource usage with throughput by instantiation of variable numbers of cores.

The MADF modelling domain is designed to enable this kind of graph-level control of the underlying processing structure. In MADF, the notions of DFG actors and edges are extended to arrays. An MASDF graph $G = V_a, E_a$ describes arrays of actors connected by arrays of arcs. An MADF graph for matrix multiplication is shown in Figure 11.18. Actor arrays are black, as opposed to single actors (or actor arrays of size 1) which are white. Arc arrays are solid, as opposed to single arcs (or arc arrays of size 1) which are dashed. The size of an actor array is quoted in angle brackets above the actor array.

In such a graph, the system designer controls parameters such as y in Figure 11.18. This is used to define the size of the *m_mult*, m_src_1, m_src_2 and *snk* actor arrays, as well as the dimensions of the tokens consumed/produced at the ports o on m_src_2, and b and o on *m_mult*. If the array of *m_mult* actors is translated to a family of cores configurable in terms of port token dimension, this allows simple graph-level control of the number of cores and token dimensions for each. However, as outlined in Section 11.3.8, cores produced via single rate dataflow synthesis have fixed port token dimensions. The technique for overcoming this restriction, is outlined next.

11.4.1 Interleaved and Block Actor Processing in MADF

Consider the case of the array of sub-matrices of m_2 input to *m_mult* in the matrix multiplication example of Figure 11.18. How may a single core be made flexible enough to implement any size of input matrix on this input, given that the pipelined core produced from an SFG description has fixed token dimensions?

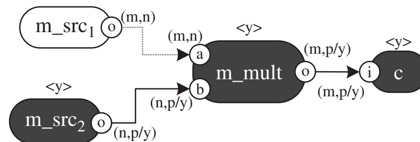


Figure 11.18 Matrix multiplication MADF

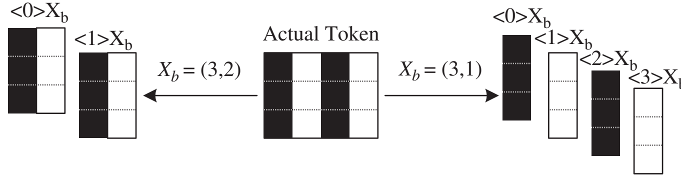


Figure 11.19 Sub-matrix decomposition for fixed token size processing

As outlined in Section 11.4, each of the **y** sub-matrices can be interpreted as a series of p column vectors, with the ith sub-matrix composed of the column vectors $i \times p/y, \ldots, ((i+1) \times p/y) - 1$ of m_2. As for the example of Section 11.4, where $y = 4$, the ith sub-matrix can be interpreted in two ways, as illustrated in Figure 11.19.

As this shows, the matrix token can be interpreted as an array of *base* tokens. If the actor to process the sub-matrix can only process the base tokens, then the entire sub-matrix may be processed using multiple iterations to process the independent base tokens. To enable this, MADF allows variable-sized arrays of actor ports, with each consuming tokens of fixed dimensions (the base token). To enable the multiple iterations of the actor to process the multiple base tokens in the actual token, MADF actors may be cyclic (Section 11.2.2), with individual firings consuming one or more base tokens through each port in the array in turn. Figure 11.20 illustrates the full, fixed token processing version of the MADF matrix multiplication problem. Note the presence of port arrays (black) with fixed token dimensions.

Having opened up the intra-token exploration space by separating the actual token processed across multiple streams transporting base tokens, further implementation exploration may be enabled. In the case where the port array is used to process a single token, *interleaved* processing of each port in the array can be implemented, i.e. a single base token is consumed through each port in turn to form the full token. In this case, the threshold of each port array element is 1. However, having opened up the token processing into a multi-stream processing problem, *block* processing is enabled, by allowing thresholds greater than 1 at each element of the port array element. At a port array, the ith element has a production/consumption vector of length p_{size} (the size of the port array) with all entries zero except the ith. These vectors exhibit a diagonal relationship, i.e. for the port array a, all entries in the consumption vector of a_0 are zero except element 0, all entries in the consumption vector for a_1 are zero except the element 1, and so forth. A generalized version of this pattern, for a port array with n elements with thresholds z is denoted by $< n > [z]$, as illustrated in Figure 11.21 for *m_mult* when $y = 3$. The value of z, the threshold on each child port indicates whether interleaved or block processing is used ($z = 1$ for interleaved, $z > 1$ for block processing).

Given this level of flexibility in the MADF structure, it is clear that given the appropriate dedicated hardware synthesis methodology, the designer can control the number of hardware components and the characteristics of each from the graph level. The most pressing concern when

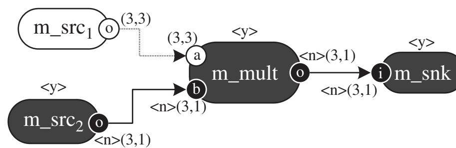


Figure 11.20 Full MADF matrix multiplication

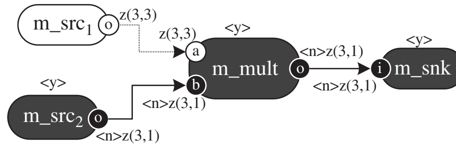


Figure 11.21 Block processing matrix multiplication

generating the embedded implementation of an MADF is the manner in which the architectural synthesis methodology can bridge the gap between the fixed semantics of the pipelined core (i.e. fixed token dimensions, threshold and number of processed streams). As outlined in this section, these restrictions are in direct contrast to the flexible nature of the MADF actors that the hardware components are to implement. An appropriate architectural synthesis methodology must be able to exploit pipelined IP cores of fixed semantics to the implementation of a MADF actor with a configuration C_b, as given in Equation (11.10), where **X**, **T** and **S** represent variable expressions of the token dimensions, thresholds and number of streams processed at each port respectively. Section 11.5 outlines how this is achieved.

$$C_b = \{T_b \ X_b \ S_b\} \tag{11.10}$$

11.5 Pipelined Core Design of MADF Algorithms

An array of MADF actors translates to an array of virtual processors (VPs) on implementation, as outlined in Figure 11.22. There are two main implementation tasks to be resolved: generation of the VP nodes and the interconnecting FIFO network. A VP is composed of three parts:

1. Control and communications wrapper (CCW)
2. Functional engine (FE)
3. Parameter bank (PB)

The CCW implements the cyclic schedule to switch the streams of data across which the VP (which represents an MADF actor) may be shared, into and out of the FE. The read unit here also implements the necessary edge FIFO buffering for the MADF network. The FE implements the

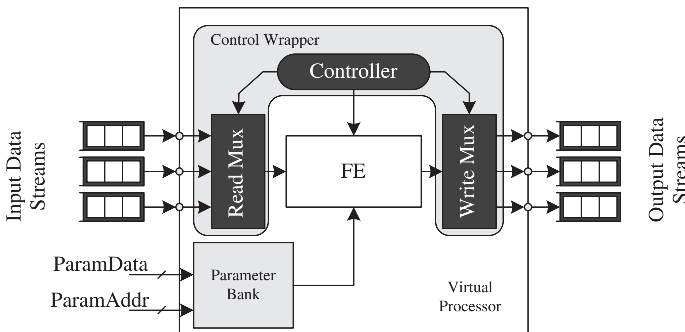


Figure 11.22 Virtual processor architecture

functionality of the node and is the core portion of the unit. It can take any architecture, but here it is a high-throughput pipelined core. The PB provides local data storage for run-time constants for the core, e.g. FIR filter tap weights. All portions of the VP are automatically generated; however, the pipelined FE core part is flexible for reuse across multiple applications and MADF actor configurations, and as such may only require creation and reuse in a core-based design strategy. Efficient generation of FIFO buffers and controllers for automatic generation of dedicated dataflow hardware is a well-researched area (e.g. Dalcolmo *et al*. 1998, Harriss *et al*. 2002, Jung and Ha 2004, Williamson and Lee 1996). Specifically, the remainder of this chapter is concerned with design of reusable FE cores.

In the case where the FE part of a VP is a pipelined core, it is designed as a *white box component* (WBC) as described in Chapter 9 (Yi and Woods 2006); the structure and behaviour of which are described here using an example two-stage finite impulse response (FIR) filter WBC, as illustrated in Figure 11.23. The WBC is composed of a computational portion and a state space and is created as described earlier in Chapter 9, but the key aspect to designing reusable, configurable cores lies in the proper design of the circuitry for the state space portion. The SFG architectural synthesis process generates a base data state space for a MADF actor, operating on base tokens. This then undergoes a series of augmentations to give the WBC a flexible internal structure which may be regularly changed without redesign to achieve regular changes in MADF actor configuration.

11.5.1 Architectural Synthesis of MADF Configurable Pipelined Dedicated Hardware

The pipelined WBC architecture resulting from SFG architectural synthesis is merely a retimed version of the original SFG algorithm, the computational resource of which must effectively be time multiplexed between each of the n elements of the input stream array, with the entire computation resource of the SFG dedicated to a single stream for a single cycle in the case of interleaved processing, and for multiple cycles in the case of block processing.

To enable interleaved processing, the first stage in the WBC state space augmentation process is lateral delay scaling. To enable interleaved sharing, the the SFG structure is k-slowed (Parhi 1999), where the delay length on every edge resulting from SFG architectural synthesis is scaled by a factor k, where in the case of interleaved processing of n input streams, $k = n$. This type of manipulation is known as *lateral delay scaling*.

In the case where block processing is required, base tokens are consumed/produced from a single port array element for a sustained number of cycles. Accordingly, the VP state space should have enough state capacity for all S streams, activating the state space associated with a single

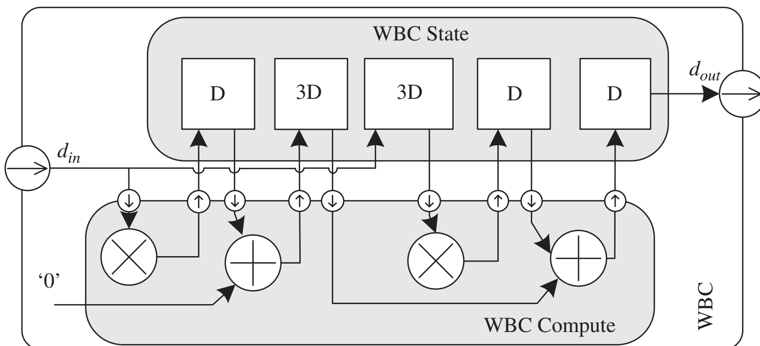


Figure 11.23 Two-stage FIR WBC

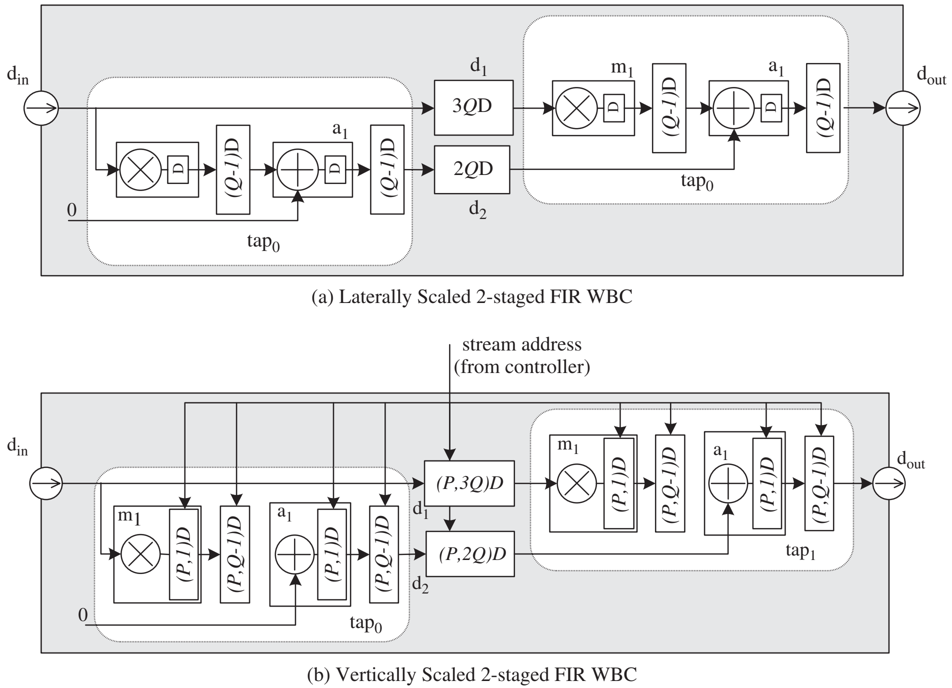
stream in turn, processing for an arbitrary number of tokens, before loading the state space for the next stream. This kind of load–compute–store behaviour is most suited to implementation as a distributed memory component, with the active memory locations determined by controller schedule. This is known here as *vertical* delay scaling, where each SFG delay is scaled into an S-element DisRAM.

Given the two general themes of lateral and vertical delay scalability, an architectural synthesis process for reusable WBC cores to allow actor and pipelined core configuration in terms of the factors in Equation (11.10), a four-stage architectural synthesis procedure has been developed.

1. *Perform MASDF Actor SFG Architectural Synthesis*. For a chosen MADF actor, C is fixed and defined as the base configuration C_b. This is converted to a SFG for architectural synthesis. The MADF actor C_b is the minimum possible set of configuration values for which the resulting pipelined architecture, the base processor P_b may be used, but, by regular alteration of the parameterized structure, the processor can implement integer supersets of the configuration. The lower the configuration values in the base, the greater the range of higher-order configurations that the component can implement. To more efficiently implement higher-order configurations, C_b can be raised to a higher value. For a two-stage FIR $C_b = \{1\ 1\ 1\}$, the WBC of the P_b is shown in Figure 11.23.
2. *Lateral Delay Scalability for Interleaved Processing*. To implement k-slowing for variable interleaved operation, the length of all delays must be scaled by a constant factor Q. All the lowest level components (adder/multipliers) are built from pre-designed cores which have fixed pipelined depths (in the case of Figure 11.23 these are all one) which cannot be altered by the designer. To enable the scaling of these delays, these are augmented with delays on their outputs to complete the scaling of the single pipeline stages to that of length Q. The resulting FIR circuit architecture for the pipelined FIR of Figure 11.23 is shown in Figure 11.24(a). The notation $(Q)D$ refers to an array of delays with dimensions $(1,Q)$. Note that all delay lengths are now a factor of Q, the lateral scaling factor, and the presence of the added delay chains on the outputs of the lowest level components. This type of manipulation is ideally suited to FPGA where long delays are efficiently implemented as shift registers (Xilinx Inc. 2005).
3. *Vertical Delay Scalability for Block Processing*. For block processing, the circuit delays are scaled by a vertical scaling factor P to allow combined interleaved/block processing. This results in arrays of delays with dimensions (P, Q). When this is applied to the circuit of Figure 11.24(a), the FIR circuit architecture shown in Figure 11.24(b) is created. Note the presence of the vertical scaling factor on all delay arrays. This kind of miniature embedded RAM-based behaviour is ideally suited to FPGA implementation, since these can implement small distributed RAMs (DisRAM) in programmable logic. These DisRAMs have the same timing profile as a simple delay (Xilinx Inc. 2005), and as such do not upset edge weights in the circuit architecture.
4. *Retime Structure to Minimize Lateral Delay Scalability*. When P_b is configured to implement a much higher order MADF actor configuration than C_b, very large delay lengths can result. To minimize these, retiming is applied to the augmented processor architecture.

11.5.2 WBC Configuration

After creation of the configurable based WBC architecture P_b, it must be configured for use with specific MADF actor configuration. To configure P_b ($C_b = \{T_b\ X_b\ S_b\}$) with pipeline period α_c



(a) Laterally Scaled 2-staged FIR WBC

(b) Vertically Scaled 2-staged FIR WBC

Figure 11.24 Scaled variants of two-stage FIR WBC

(Parhi 1999) to implement a higher-order MADF actor P, where X is an n-dimensional token of size $x(i)$ in the ith dimension, the following procedure is used.

1. Determine lateral scaling factor Q by Equation (11.11)

$$Q = \left\lceil \frac{1}{\alpha_\text{c}} \prod_{i=0}^{n-1} \frac{xi}{x_\text{b}i} \right\rceil \tag{11.11}$$

2. Scale all edge delay lengths by a factor Q. Scale primitive output delays to length $(Q-1) \times L$ where $\boldsymbol{L}$ is the number of pipeline stages in the primitive.
3. Scale all delays vertically by a vertical scaling factor P as given by Equation (11.12)

$$P = \frac{S}{S_\text{b}} \tag{11.12}$$

11.6 System-level Design and Exploration of Dedicated Hardware Networks

This section outlines two examples, namely a normalized lattice filter (NLF) and a fixed beamformer, in order to demonstrate many of the features outlined in the previous sections.

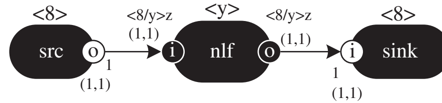


Figure 11.25 Eight-stage NLF MADF graph

11.6.1 Design Example: Normalized Lattice Filter

To demonstrate the effectiveness of this architectural synthesis and exploration approach, it is applied to an eight-stage NLF design problem given in (Parhi 1999). The MADF graph is shown in Figure 11.25, with the NLF actor, the only part implemented in dedicated hardware as part of this design example.

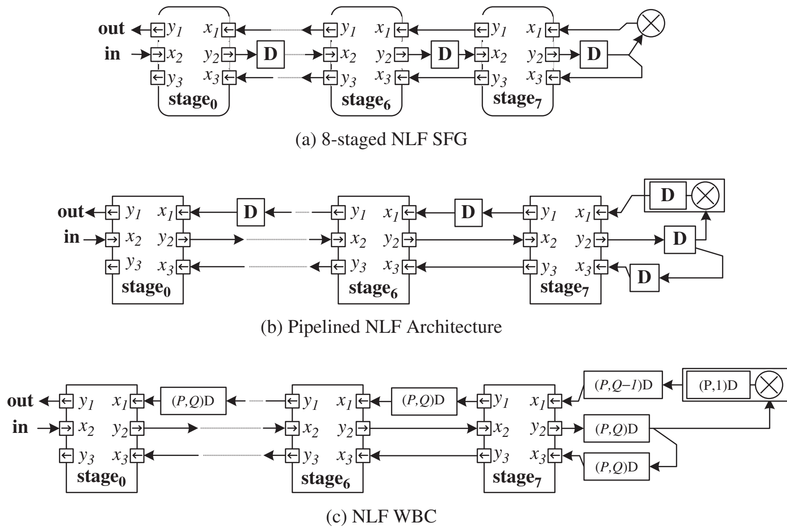
As Figure 11.25 shows, the *src* and *sink* arrays generate an array of eight scalar tokens which are processed by the NLF array. The designer controls the size of the NLF actor array by manipulating the variable y on the graph canvas. This, in turn, determines n, the size of the port array of each element of the NLF actor array. To test the efficiency of this MADF synthesis and exploration approach, the SFG architectural synthesis capability for P_b is limited to only retiming (i.e. advanced architectural explorations such as folding/unfolding are not performed), placing the emphasis for implementation optimization entirely on the MADF design and exploration capabilities. The P_b operates on scalar tokens with $C_b = \{1b\ 1b\ 1\}$ to maximize SFO flexibility by maximizing the number of achievable configurations. The target device is the smallest possible member of the Virtex-II Pro™ family which can support the implementation. This enables two target device specific design rules for efficient synthesis:

1. if $(P, Q) = (1, 1) D_{\text{type}} = \text{FDE}$
2. if $P > 1$, $D_{\text{type}} = \text{DisRAM (RAM16x1s)}$, otherwise $D_{\text{type}} = \text{SRL16} + \text{FDE}$.

The SFG of an 8 stage NLF with $C_b = \{1b\ 1b\ 1\}$ is shown in Figure 11.26(a), with the SFG of the NLF stage shown in Figure 11.27(a). If the lowest-level components (adders and multipliers) from which the structure is to be constructed are implemented using single-stage pipelined black box components (a common occurrence in modern FPGA), then a particular feature of the NLF structure is the presence of 36 recursive loops in the structure, with the critical loop (Parhi 1999) occurring when two pipelined stages are connected. For single-stage pipelined adders and multipliers, this has a pipeline period, (α), of 4 clock cycles. Hence by Equation (11.11), $Q = \frac{x_i}{4 \times x_b}$.

The base processor P_b is created via hierarchical SFG architectural synthesis (Yi and Woods 2006), and produces the pipelined architecture of Figure 11.26(b), with the architecture of each stage as shown in Figure 11.27(b). After lateral and vertical delay scaling and retiming, the NLF and stage WBC architectures are as shown in Figure 11.26(c) and Figure 11.27(c) respectively.

Synthesis of the given architecture for three different values of y has been performed. ${}^{8}_{1}$*BS-NLF*, ${}^{2}_{4}$*BS-NLF* and ${}^{1}_{8}$*BS-NLF* are the structures generated when y is 8, 2 and 1 respectively and each VP performs interleaved sharing over the impinging data streams, whilst results for a single VP processing a 68 element vector (${}^{68}_{1}$*BS-NLF*), are also quoted to illustrate the flexibility of the WBC architectures. A block processing illustration of 16 streams of four-element vector tokens (${}^{1}_{4}$*BS-NLF*$_{16}$) is also quoted in Table 11.4. These results illustrate the effectiveness of this approach for core generation and high-level architecture exploration. Transforming the MASDF specification by trading off number of actors in the family, token size per actor, and number of functions in the MASDF actor cyclic schedule, has enabled an effective optimization approach without redesign.

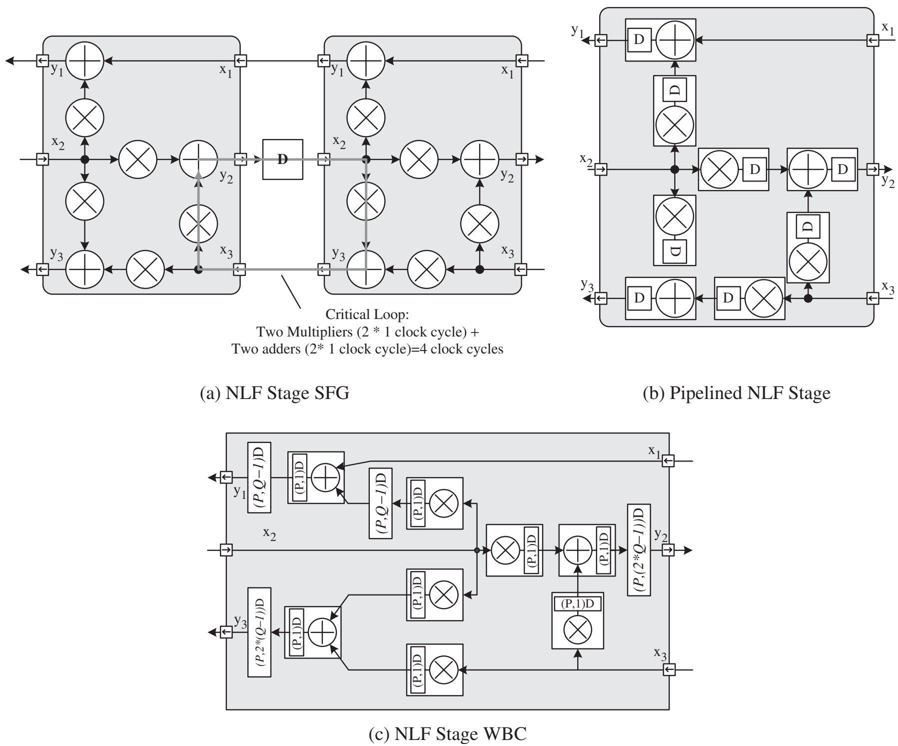


(a) 8-staged NLF SFG

(b) Pipelined NLF Architecture

(c) NLF WBC

Figure 11.26 NLF SFG, pipelined architecture and WBC



(a) NLF Stage SFG

(b) Pipelined NLF Stage

(c) NLF Stage WBC

Figure 11.27 NLF stage SFG, pipelined architecture and WBC

Table 11.4 NLF post Place and route synthesis results on Virtex-II Pro FPGA

Configuration	Slices	Logic				mult18	Throughput
		LUTs	SRL	DisRAM	FDE		(Msamples/s)
${}^{8}_{1}$*BS-NLF*	2784	1472			4840	312	397.4
${}^{2}_{4}$*BS-NLF*	670	368			1210	78	377.9
${}^{1}_{8}$*BS-NLF*	442	186	207		801	39	208.6
${}^{68}_{1}$*BS-NLF*	442	186	207		801	39	208.6
${}^{1}_{4}$*BS-NLF*$_{16}$	508	188	7	576	105	39	135.8

The initial implementation ($y = 8$, ${}^{8}_{1}$*BS-NLF*) created an eight-element VP array. Given the large number of multipliers (mult18) required for implementation, the smallest device on which this architecture can be implemented is an XCV2VP70. However, given the pipeline period inefficiency in the the original WBC architecture, reducing y to 2 produces two four-element vector processors (${}^{2}_{4}$*BS-NLF*) with almost identical throughput, enables a significant reduction in required hardware resource with little effect on throughput. This amounts to a throughput increase by a factor 3.9 for each VP with no extra hardware required in the WBC. The large reduction in required number of embedded multipliers also allows implementation on a much smaller XC2VP20 device. Decreasing y still further to 1, produces a single eight-element vector processor (${}^{1}_{8}$*BS-NLF*). Whilst throughput has decreased, a significant hardware saving has been made. The NLF array can now be implemented on a smaller XC2VP7 device.

This example shows that the MADF synthesis approach can achieve impressive implementation results via simple system-level design space exploration. Using a single pipelined core, this approach has enabled highly efficient architectures (3.9 times more efficient than one-to-one mappings) to be easily generated, in a much simpler and more coherent manner than in SFG architectural synthesis. Furthermore, by manipulating a single DFG-level parameter, this design example can automatically generate implementations with wildly varying implementation requirements, offering an order of magnitude reduction in device complexity. This illustrates the power of this approach as a system-level, core-based design flow with highly efficient implementation results and rapid design space exploration capabilities.

11.6.2 Design Example: Fixed Beamformer System

Beamforming provides an effective and versatile method of spatial filtering for radar, sonar, biomedical and communications applications (Haykin 1986). A beamformer is typically used with an array of sensors which are positioned at different locations so that they are able to ‘listen’ for a received signal by taking spatial samples of the received propagating wave fields. The structure of a fixed beamformer (FBF) is shown in Figure 11.28.

The FBF samples complex data impinging on N sensors and performs independent FIR filtering and scaling on the output of each of the N samples by individual elements of a constant weighting vector to steer the beamformer response spatially toward a particular target. The scaled values are all then summed. For a target, the average power at the output of the beamformer is maximized when the beamformer is steered toward the target. The structure of the FBF system is highly regular, and can be exploited by the MADF modelling domain to provide wide-ranging and effective design space exploration. The MADF graph of the FBF algorithm is shown in Figure 11.29.

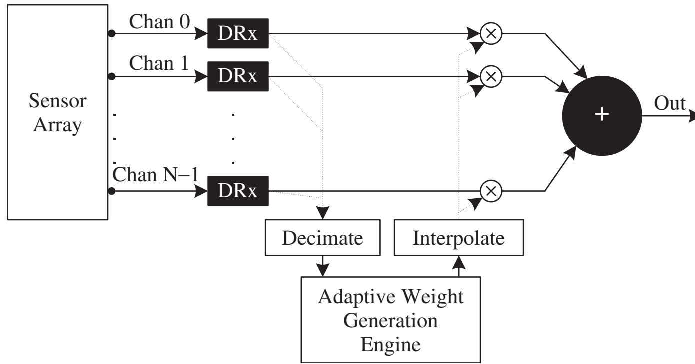


Figure 11.28 Fixed beamformer overview

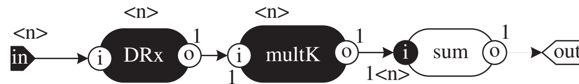


Figure 11.29 Fixed beamformer MADF graph

The MADF graph consists of an array of N inputs, one for each sensor in the array. This is tightly correlated with the number of members in the *DRx* and *multK* actor families, as well as the size of the port family i on the *sum* actor (again a port family is denoted in black). Hence, by altering the value of N, parameterized control of the algorithm structure is harnessed for a variable number of sensors. By coupling the implementation structure tightly to the algorithm structure, this gives close control of the number of *DRx* and *multK* cores in the implementation.

For the purposes of this design example, $N = 128$ and the design process targets a Xilinx Virtex II Pro 100 FPGA. The core library consists only of complex multiplication, addition and sum cores, and hence the entire system is to be composed from these. The lengths of the *DRx* filters is taken as 32 taps. Given that this structure then requires 16896 multipliers, and it is desirable to utilize the provided 18 bit multipliers on the target device (of which only 444 are available), this presents a highly resource-constrained design problem. The approach here offers two ways to help counteract this problem: architectural manipulation when the functionality has been implemented already, and processing of multiple streams of data in each VP.

To enable the exploration of the number of channels processed by each core in the implementation, each actor must be able to process multiple channels in the MADF algorithm. This is enabled using the MADF structure of Figure 11.30. Here, a second parameter, M has been introduced to denote the number of actors used to process the N channels of data. Note that the ports on the *DRx* and *multK* actors are now both families of size M to denote the sharing of the actor amongst N/M data streams processed in a cyclic fashion (McAllister *et al.* 2006). On synthesis, a wide range of synthesis options are available for the FBF dedicated hardware system on a chosen device, with an accompanying wide range of real-time performance capabilities and resource requirements, and these are summarized in Table 11.5. The breakdown of the proportion of the programmable logic (LUT/FDE) by VP function (WBC, PB or CCW) is given in Table 11.6

An initial implementation consisting of a single core can process a 128-element vector (i.e. interleave shared across the 128 input streams); increasing the value of M by 2 and 4 can produce increases in throughput by factors of 2.2 and 4.3, respectively; it should be noted

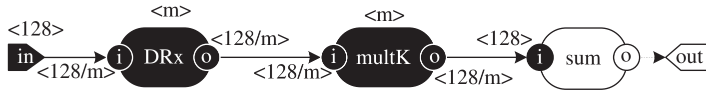


Figure 11.30 Fixed beamformer MADF graph

Table 11.5 FBF post-place and route synthesis results on Virtex II Pro FPGA

$M(i)$	Logic LUT/SRL/DisRAM/FDE	mult18 (%)	*mux* F5/F6/F7/F8(%)	Throughput (MSamples/s)
1(i)	3493/16128/8448/5790 (8)/(37)/(19)/(6)	99 (22)	528/256/128/64 (1)/(1)/(1)/(1)	1.45
2(i)	4813/16128/8448/10844 (11)/(37)/(19)/(11)	198 (45)	512/256/128/64 (1)/(1)/(1)/(1)	3.18
4(i)	8544/16128/8448/21576 (19)/(37)/(19)/(22)	396 (89)	528/256/128/64 (1)/(1)/(1)/(1)	6.19
1(b)	3490/0/24576/5278 (8)/(0)/(56)/(5)	99 (22)	528/256/128/64 (1)/(1)/(1)/(1)	1.45
2(b)	4812/0/24576/8892 (11)/(0)/(56)/(9)	198 (45)	528/256/128/64 (1)/(1)/(1)/(1)	3.51
4(b)	8554/0/24576/17672 (19)/(0)/(56)/(18)	396 (89)	528/256/128/64 (1)/(1)/(1)/(1)	1.45

Table 11.6 FBF implementation resource breakdown

$M(i)$	LUT			FDE		
	%WBC	%CCW	%PB	%WBC	%CCW	%PB
1(i)	3.7	31.3	6.5	1.3	0	98.7
2(i)	3.6	28.7	69.5	0.9	0	99.1
4(i)	3.2	25.5	71.3	0.3	0	99.7
1(b)	3.7	31.3	6.5	1.3	0	98.7
2(b)	3.6	28.7	69.5	1.0	0	99.0
4(b)	3.2	25.5	71.3		0.3	99.7

that the architectures used for core sharing amongst multiple streams exhibit minimal resource differences. This is a direct result of the abstraction of the core architectures for target portability. Whilst the overheads in terms of LUTs (which may be configured as 16 bit shift registers (SRLs) or DisRAMS for the WBC wrapping in the VP is expensive (up to 35% overhead) the major part of this is required entirely for storage of on-chip filter tap and multiplier weights in the SFO parameter banks. This storage penalty is unavoidable without exploiting on-chip embedded BlockRAMs. In addition, the overhead levels decrease with increasing values of M since the number of tap weights remains constant independent of M. The CCW incurs little LUT overhead, instead exploiting the embedded *muxF5*, *muxF6*, *muxF7* and *muxF8* multiplexers of the FPGA (Xilinx Inc. 2005) to implement the switching. These are not used at all anywhere else in

the design and hence are plentiful. Finally, it should be noted that all the cores in the system are 100% utilized, depending on input data.

11.7 Summary

This chapter has described techniques to enable system-level design and optimization of dedicated hardware networks in FPGA. It is important to note that in industrial system design flows, dedicated hardware generally forms a small part of a final embedded DSP system, and as such its design, integration and reuse should lend itself well to industrial heterogeneous system design flows.

Popularly, rapid DSP system implementation approaches are based on dataflow-centric approaches. Whilst rapid implementation of pipelined DSP cores is in itself a dataflow design flow, the final product of the process is rigid and inflexible for integration, reuse and manipulation, along with software parts of the system. For such approaches to become popularly accepted, this situation must be rectified.

A number of techniques have been outlined to bridge the gap between what system designers require and what dedicated hardware permits. By accepting that flexibility is imperative in the resulting dedicated hardware, this section has shown that dedicated hardware creation, manipulation and reuse can be effectively merged into a heterogeneous system design flow.

The use of MADF as a modelling approach for DSP systems helps encapsulate the required aspects of system flexibility for DSP systems, in particular the ability to exploit data-level parallelism, and control how this influences the implementation. This has been shown to be an effective approach; for a NLF filter design example, impressive gains in the productivity of the design approach were achieved. This included an almost four-fold increase in the efficiency of the implementation via simple transformations at the DFG level, negating the need for complex SFG architectural manipulations. Otherwise, this approach has proven effective at rapid design space exploration, producing NLF implementations of varying throughput and drastically different physical resource requirements (an order of magnitude variation in device complexity) simply by manipulating a single parameter at the graph level. Furthermore, in a FBF design example, the effectiveness of this approach has been demonstrated by enabling rapid design space exploration, producing a variety of implementations for a specific device via manipulation of a single DFG parameter.

References

Bhattacharyya S, Murthy P and Lee E (1999) Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing* **21**(2), 151–166.

Bilsen G, Engels M, Lauwereins R and Peperstraete J (1996) Cyclo-static dataflow. *IEEE. Trans Signal Processing* **44**(2), 397–408.

Dalcolmo J, Lauwereins R and Ade M (1998) Code generation of data dominated DSP applications for FPGA targets *Proc. 9th International Workshop on Rapid System Prototyping*, pp. 162–167.

Gajski D, Vahid F, Narayan S and Gong J (1994) *Specification and Design of Embedded Systems*. Prentice Hall.

Grötker T, Liao S, Swan S and Martin G (2002) *System Design with System C*. Kluwer.

Harriss T, Walke R, Kienhuis B and Deprettere EF (2002) Compilation from matlab to process networks realised in fpga. *Design Automation for Embedded Systems* **7**(4), 385–403.

Haykin S (1986) *Adaptive Filter Theory*. Prentice Hall, Englewood Cliffs, NJ.

Jung H and Ha S (2004) Hardware synthesis from coarse-grained dataflow specification for fast HW/SW cosynthesis *Proc. International Conference on Hardware/Software Codesign and System Synthesis*, pp. 242–29.

Kahn G (1974) The Semantics of a Simple Language for Parallel Programming *Proc. IFIP Congress*, pp. 471–475.

Kalavade A and Lee E (1997) The extended partitioning problem: hardware/software mapping, scheduling and implementation-bin selection. *Design Automation for Embedded Systems* **2**(2), 125–163.

Keutzer K, Malik S, Richard Newton S, Rabaey JM and Sangiovanni-Vincentelli A (2000) System level design: orthogonolization of concerns and platform-based design. *IEEE Trans. CAD* **19**, 1523–1543.

Lee E (1991) Consistency in dataflow graphs. *IEEE Trans. Parallel and Distributed Systems* **2**(2), 2233–2235.

Lee E and Messerschmitt D (1987a) Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers* **C-36**(1), 24–35.

Lee E and Messerschmitt D (1987b) Synchronous data flow. *Proc. IEEE* **75**(9), 1235–1245.

Lee E and Sangiovanni-Vincentelli A (1998) A framework for comparing models of computation. *IEEE. Trans. CAD* **17**(12), 1217–1229.

Lee EA (1993a) Multidimensional streams rooted in dataflow. *IFIP Transactions; Vol. A-23: Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pp. 295–306.

Lee EA (1993b) Representing and exploiting data parallelism using multidimensional dataflow diagrams *Proc. 1993 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP-93)*, pp. 27–30.

Lee EA and Parks TM (1995) Dataflow process networks. *Proc. IEEE* **85**(3), 773–799.

McAllister J, Woods R, Walke R and Reilly D (2006) Multidimensional DSP core synthesis For FPGA. *Journal of VLSI Signal Processing* **43**(2/3), 207–221.

Murthy PK and Lee EA (2002) Multidimensional synchronous dataflow. *IEEE Trans. Signal Processing* **50**(8), 20647–20799.

Najjar WA, Lee EA and Gao GR (1999) Advances in the dataflow computational model. *Parallel Computing* **25**(4), 1907–1929.

Parhi KK (1999) *VLSI digital signal processing systems : design and implementation*. John Wiley and Sons, Inc., New York.

Parks T, Pino J and Lee E (1995) A comparison of syncronous and cyclo-Static dataflow. *Conference Record of the 29th Asilomar Conference on Signals, Systems and Computers*, pp. 204–210.

Rowson JA and Sangiovanni-Vincentelli A (1997) Interface-based design *Proc. 34th Design Automation Conference*, pp. 178–183.

Sriram S and Bhattacharyya S (2000) *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker.

VSIA (2007) Vsia quality ip metric. Web publication downloadable from http://www.vsia.org/documents/.

Williamson M and Lee E (1996) Synthesis of parallel hardware implementations from synchronous dataflow graph specifications *Proc. 30th Asilomar Conference on Systems, Signals and Computers*, pp. 1340–1343.

Xilinx Inc. (2005) Virtex-ii pro and virtex-ii pro x platform fpgas: Complete data sheet. Web publication downloadable from http://www.xilinx.com.

Yi Y and Woods R (2006) Hierarchical synthesis of complex dsp functions using iris. *IEE Trans. Computer Aided Design* **25**(5), 806–820.

12

Adaptive Beamformer Example

Chapter 8 covered techniques to create efficient circuit architectures for SFG-based DSP systems. In this description, it was clear that many of the implementations could be created in a generic fashion with levels of pipelining used as a control parameter. Chapter 10 indicated that development of this generic architecture could be used across a wide range of applications if the parameterizable features could be used to drive efficient implementations across this range of parameters. This can only be achieved by identifying the key parameters of the DSP core functionality and then cleverly deriving a scalable architecture that will allow these parameters to be altered, whilst still producing linearly scaled performance. This requires a combination of high-level design optimizations and use of the techniques highlighted in Chapter 8.

This chapter is dedicated to the development of a QR-based IP core for adaptive beamforming. This example covers a number of stages, from the development of the mathematical algorithm through to the design of a scalable architecture. Focus is given to the techniques used to take the original architecture, then map and fold it down onto an efficient and scalable implementation, meeting the needs of the system requirements. Issues such as parameterizable timing and control management are also covered and how this relates to the earlier techniques.

The chapter is organized as follows. Section 12.1 gives an introduction to adaptive beamforming and the generic design process is covered in Section 12.2. The adaptive beamforming application is outlined in Section 12.3 and the development of the QR-based algorithm covered in Section 12.4. Algorithm to architecture procedures are covered in Section 12.5 and then applied to produce the efficient architecture design given in Section 12.6. A series of different architectures are developed. These are then used to develop the generic architecture which is given in Section 12.7. The details of how the architecture is retimed to cope with processors with detailed timing considerations are given in Section 12.8, leading to the parameterized QR architecture, highlighted in Section 12.9. In Section 12.10, the issues of generic control for this architecture is covered. Finally, a beamformer application is described in Section 12.11 and followed by conclusions.

12.1 Introduction to Adaptive Beamforming

Adaptive beamforming is a form of filtering whereby input signals are received from a number of spatially separated antennae, referred to as an antennae array. Typically its function is to suppress signals from every direction other than the desired 'look direction' by introducing deep nulls in the beam pattern in the direction of the interference. The beamformer output is a weighted linear combination of input signals from the antennae array, represented by complex numbers, therefore

FPGA-based Implementation of Signal Processing Systems R. Woods, J. McAllister, G. Lightbody and Y. Yi
© 2008 John Wiley & Sons, Ltd

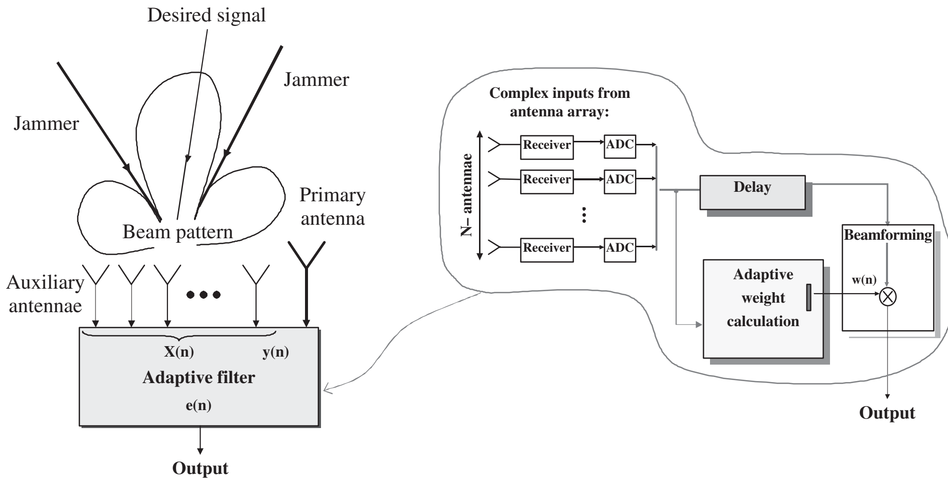


Figure 12.1 Diagram of an adaptive beamformer for interference cancelling. Reproduced from Lightbody *et al*., © 2003 IEEE

allowing an optimization both in amplitude and phase due to the spatial element of the incoming data.

Figure 12.1 illustrates an example with one primary antenna and a number of auxiliary antennae. The primary signal constitutes the input from the main antennae, which has high directivity. The auxiliary signals contain samples of interference, threatening to swamp the desired signal. The filter eliminates this interference by removing any signals in common with the primary input signal. The input data from the auxiliary and primary antennae is fed into the adaptive filter, shown in Figure 12.1, from which the weights are calculated. These weights are then applied on the delayed input data to produce the output beam, as depicted in Figure 12.1. It is the choice and development of an algorithm for adaptively calculating the weights that is the focus of this chapter.

There are a range of applications for adaptive beamforming from military radar applications to communications and medical applications, (Athanasiadis *et al*. 2005, Baxter and McWhirter 2003, Choi and Shim 2000, de Lathauwer *et al*. 2000, Hudson 1981, Shan and Kailath 1985, Wiltgen 2007). Due to the possible applications for such a core, this chapter will investigate the development of an IP core to perform the key computation found in a number of such adaptive beamforming applications.

12.2 Generic Design Process

In the development of a new hardware core, a series of design stages may be followed. Figure 12.2 gives a summary of a typical design process applied in the development of a single-use implementation. It also gives the additional considerations required in generic IP core design, as highlighted in Chapter 10. A number of key issues are addressed.

The process begins with a *detailed specification* of the problem and the purpose of the design. At this point, consideration may be given toward employing *design for reuse* strategies to develop a generic end product. Considerations include the initial extra cost in terms of money and time in the development of a generic core, so it is essential that this cost will be more than recouped if

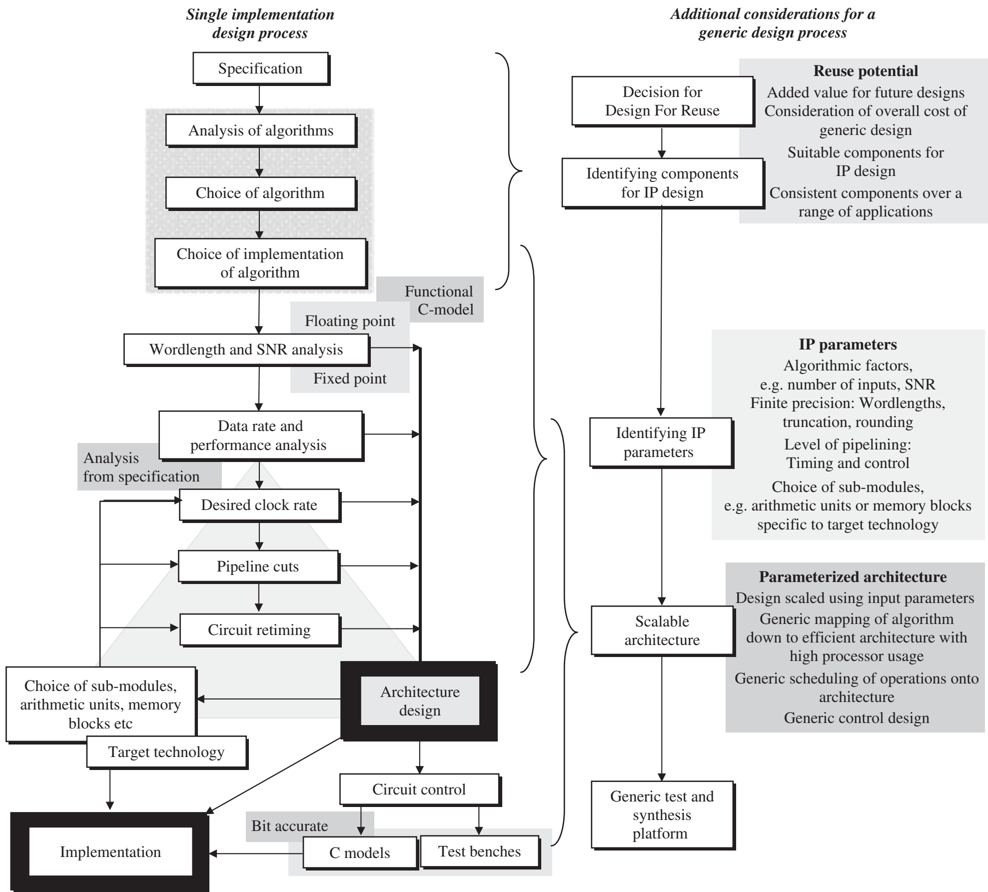


Figure 12.2 Generic design process. Reproduced from Lightbody *et al.*, © 2003 IEEE

the IP core is used in future designs. Is the development applicable over a range of applications or is it a one-off requirement?

Analysis is then required to determine the most suitable algorithm. The choice of this algorithm is key as successful hardware implementation requires a detailed understanding of how the mathematical functionality is implemented. This has impact on the overall performance of a circuit, in terms of area, critical path, and power dissipation.

Identification of the *core functionality* as there may only be a number of key components that are suitable to be implemented as IP cores. These are functions that will transfer from application to application, so it is imperative to determine expected variations in specification for future applications. Are all these variables definable with parameters within the generic design, or would other techniques be required to create the flexibility of the design?

Wordlength analysis to determine fixed-/floating-point arithmetic and consideration of rounding and truncation within the design.

Details of architecture design which will determine possible clock rates and area requirements. This will depend on the target technology or specific FPGA device, influence the choice of submodules involving analysis of levels of parallelism and pipelining. It is an interlinked loop, as depicted in Figure 12.2, with each factor influencing a number of others. All these factors have an influence over the final architecture design and it is a multidimensional optimization with no one parameter operating in isolation.

Determination of parameters for the generated architectures. Within a generic design, different allowable ranges may be set on the parameters defining the generated architectures. For example, different wordlength parameters will have a knock-on effect on the level of pipelining required to meet certain performance criteria.

Generic design choices could be included for a range of target implementations, e.g. parameters to switch between ASIC and FPGA specific code. Even within a certain implementation platform, there should be parameters in place to support a range of target technologies or devices, so to make the most of their capabilities and the availability of on-board processors or arithmetic units.

Refinement of architecture solution to meet the performance criteria, but at a reduced area cost. This includes application of the folding techniques outlined in Chapter 8, but the key mechanics of a successful generic design also require the development of scalable control circuitry and scalable scheduling of operations. Generating an architecture to meet the performance criteria of a larger design is one thing, but developing the generic scheduling and control of such a design is of a different level in complexity.

Software modelling of the algorithm is essential in the design development, initially for verification and to analyse the finite precision effects. Such a model then forms the basis for further development and implementation of the hardware architecture involving creation of test data for validation of the HDL code and synthesized netlist. For the generic IP core the software modelling forms an important part of the reuse design process as analysis is still required from the outset to determine the desired criteria for the implementation such as SNR and data wordlengths for the new applications.

12.3 Adaptive Beamforming Specification

The specification for this design is to have a generic core that can be redeveloped quickly for future adaptive beamforming applications. The process of designing such a generic architecture adds to the development time, and the future potential should be considered up front before deciding to apply a design for reuse methodology. However, gains can be made if such a generic core has reuse potential, providing for powerful and useful design libraries.

Adaptive beamforming is a general algorithm applicable in a range of applications, from medical separation of signals to military radar applications. The key factor for development of a generic design is to determine the key component within a range of adaptive beamforming applications that would be consistent to some degree and could therefore be suitable to develop as an IP core. To widen the potential of the core, it will need to be able to support a varied range of specifications dealing with issues such as the following.

Number of Inputs

The beamformer design will need to support a varied number of auxiliary and primary inputs, as depicted in Figure 12.3. The weights are calculated for a block of the input data coming from N antennae (generally, only a proportion of this input data is needed, but there should be at least $2N$ data samples used from each block). These weights are then applied to the same input data to generate the beamformer output for that block. For the final design, a more efficient post-processor is developed to extract the weights such as that described in Shepherd and McWhirter (1993).

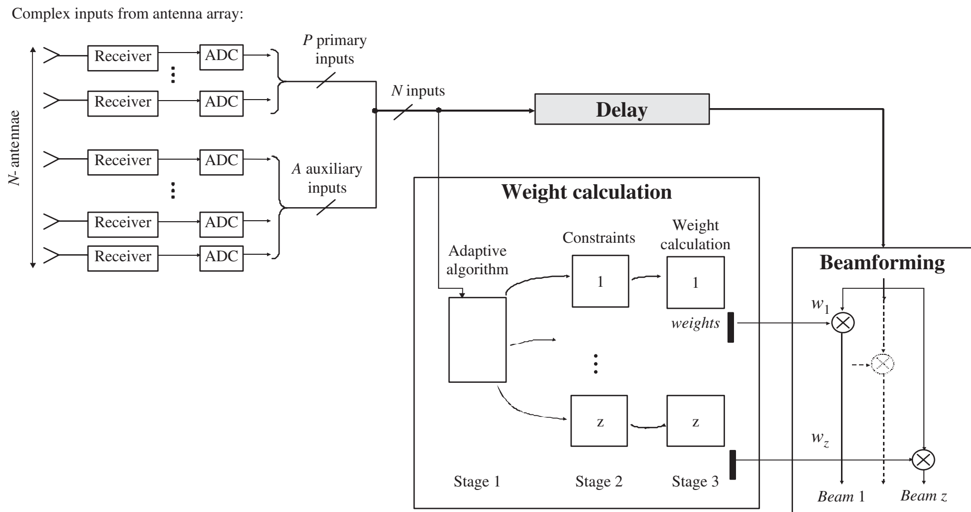


Figure 12.3 Multiple beam adaptive beamformer system. Reproduced from Lightbody *et al.*, © 2003 IEEE

Supporting a Range of FPGA Devices and/or ASIC Technologies

By including some additional code and parameters, the same core design can be re-targeted to a different technology. Doing this could enable a design to be prototyped on FPGA before targeting to ASIC. It would also allow for low-yield implementations that would not warrant the ASIC design overhead. Likewise, there is a great benefit in the ability to quickly redevelop the core for emerging ASIC foundries.

Ability to Support a Range of Performance Criteria

The variation in adaptive beamformer applications creates a wide span of desired features. For some applications, for example, mobile communications, power consideration and chip area could be the driving criteria for the device. For others, a high data rate system could be the primary objective.

Scalable Architecture

To create the flexibility needed to support such a wide range of design criteria, a scalable architecture needs to be developed that can increase the level of physical hardware to match the needs of the specification. Some key points driving the scalable architecture are

- desired data rate
- area constraints
- clock rate constraints
- power constraints

Clock Rate Performance

The required clock rate for the system is dependent on the architecture design and the target technology. Specifying the system requirements enables the designer to make a choice regarding

the target technology and facilitates compromise with other performance criteria such as power and area.

Wordlength

Different applications will require different wordlengths, therefore a range of wordlengths should be supported.

Level of Pipelining

The desired clock rate may rely on pipelining within the design to reduce the critical path. Giving a choice of pipelining within the submodules of the design will have a great influence over performance.

These values will form the basis from which to develop the adaptive beamformer solution from the generic architecture. The surrounding software models and testbenches should include the same level of scalability so to complete the parameterization process.

The remainder of this chapter will describe the design stages in the development of a generic adaptive beamformer core suitable for parameterization and fast prototyping. A summary is presented, showing the transition from problem to mathematical algorithm. From this a suitable solution is determined and an architecture is derived. A focus is given for the later as a design for reuse methodology is followed with the aim of developing a generic core, forming a key component of the full adaptive beamforming application.

12.4 Algorithm Development

The aim of an adaptive filter is to continually optimize itself according to the environment in which it is operating. A number of mathematically and highly complex algorithms exist to calculate the filter weights according to an optimization criterion. Typically, the target is to minimize an error function, which is the difference between a desired performance and the actual performance. Figure 12.4 highlights this process.

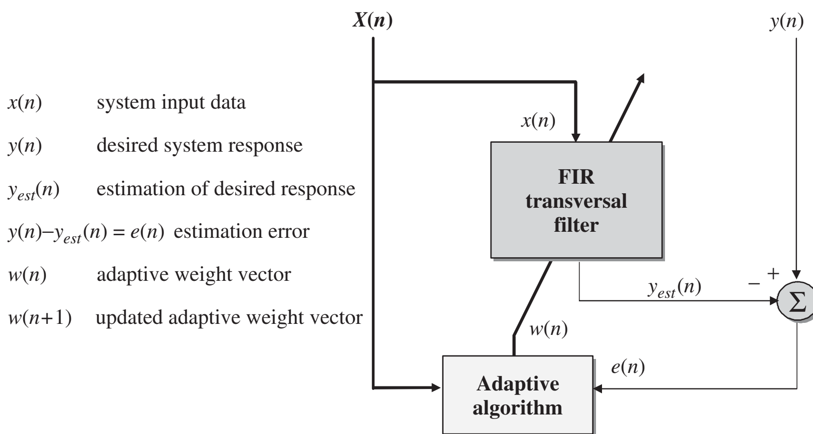


Figure 12.4 Adaptive filter system

A great volume of research has been carried out into different methods for calculating the filter weights. *Adaptive Filter Theory* by Haykin (2002) gives a well-rounded introduction to adaptive filtering. The possible algorithms range in complexity and capability, requiring detailed analysis to determine a suitable algorithm. However there is no distinct technique in finding the optimum adaptive algorithm for a specific application. The choice comes down to a balance of the range of characteristics defining the algorithms, such as:

1. Rate of convergence, i.e. the rate at which the adaptive algorithm comes within a tolerance of an optimum solution
2. Steady-state error, i.e. the proximity to an optimum solution
3. Ability to track statistical variations in the input data
4. Computational complexity
5. Ability to operate with ill-conditioned input data
6. Sensitivity to variations in the wordlengths used in the implementation

Two methods for deriving recursive algorithms for adaptive filters use Wiener filter theory and the method of least-squares (LS), resulting in the LMS and RLS algorithms respectively which have been covered in Chapter 2. The key issue concerning the choice of these algorithms is complexity versus performance.

12.4.1 Adaptive Algorithm

It is widely accepted that RLS solutions offer superior convergence rates to the LMS solutions under stationary environments. A RLS solution would therefore be expected to react faster under non-stationary conditions; however, this is not a straightforward comparison (Eleftheriou and Falconer 1986, Eweda 1998, Eweda and Macchi 1987, Haykin 2002, Kalouptsidis and Theodoridis 1993). General observations made from these references include:

In general, the RLS algorithm is preferable over the LMS algorithm when tracking performance and speed of convergence is critical, especially in situations where there is a high-to-medium SNR (Kalouptsidis and Theodoridis 1993).

The convergence rate of the LMS algorithm is governed by the choice of step size μ, where increasing μ accelerates the convergence. However, this does not compensate fully for the slower rate as the algorithm loses the benefits of noise smoothing at the steady state when μ is large. There are normalized versions that allow for a greater step size at the start, which is then reduced as the algorithm approaches its steady state (Bitmead and Anderson 1980, Kalouptsidis and Theodoridis 1993, Morgan and Kratzer 1996).

The convergence rate of the RLS algorithm is independent of the spread of eigenvalues within the input correlation matrix. This is not the case for the LMS algorithm, as when the eigenvalue spread of the correlation matrix is great then the convergence is rather slow, (Eweda and Macchi 1987), i.e. the eigenvalues effectively form a convergence time constant and determine the value of the correction size μ, which ensures stability (Eleftheriou and Falconer 1986).

The memory of the adaptive algorithm governs the rate of convergence. The RLS algorithm incorporates a forgetting factor λ, which assigns greater importance to more recent data. It takes a value between 0 and 1 and acts to determine a window of data on which the LS solution is carried out. With larger values of λ, the window length will be longer and will possess a longer memory. Bringing λ further away from 1 shortens the memory and enables the algorithm to track the statistical changes within the data. However, λ also governs the rate of convergence and the steady-state error. In a stationary environment the best steady-state performance results

from slow adaptation where λ is close to 1. Conversely, smaller values of λ result in faster convergence but greater steady state error. This is similar to the memory-convergence relationship existing within the LMS algorithm, which is determined by the step size μ.

The optimum choice comes down to a fine balance of convergence rate, steady-state error, tracking ability, numerical stability, and computational complexity. The main hindrance of the RLS algorithm has been its computational complexity; however with technological growth the use of RLS in real-time applications is becoming feasible. The RLS solution was chosen over the LMS solution for the example presented here due to its superior convergence rates and reduced sensitivity to ill-conditioned data.

12.4.2 RLS Implementation

As outlined in Section 2.7.4, the standard RLS algorithm requires the explicit computation of the correlation matrix, $\phi(n) = X^{\mathrm{T}}(n)X(n)$. This is an intensive computation that has the effect of squaring the condition number of the problem, causing a negative effect on the required wordlength for stability in finite wordlength systems. The weights can be found in a more stable manner, avoiding both the computation of the correlation matrix and its inverse by using QR decomposition, a form of orthogonal triangularization with good numerical properties.

The choice of implementation uses QR decomposition (QR-RLS) as the central algorithm for adaptively calculating the filter weights, (Gentleman and Kung 1981, McWhirter 1983).

12.4.3 RLS Solved by QR Decomposition

The $P \times N$ dimensioned data matrix, $X(n)$, is decomposed into an $N \times N$ dimensioned upper triangular matrix, $R(n)$, through the application of a unitary matrix, $Q(n)$, such that:

$$Q(n)X(n) = \begin{bmatrix} R(n) \\ O \end{bmatrix} \tag{12.1}$$

where O is a zero matrix resulting if $N < P$. Since $Q(n)$ is a unitary matrix, then:

$$\phi(n) = X^{\mathrm{T}}(n)X(n) = X^{\mathrm{T}}(n)Q^{\mathrm{T}}(n)Q(n)X(n) = R^{\mathrm{T}}(n)R(n) \tag{12.2}$$

The triangular matrix, $R(n)$, is the Cholesky (square root) factor of the data correlation matrix $\phi(n)$. Since $Q(n)$ is unitary then the original system equation may be expressed as:

$$\|J(n)\| = \|Q(n)e(n)\| = \left\| \underbrace{Q^T(n)X(n)}_{R(n)} W_{LS}(n) + \underbrace{Q^T(n)y(n)}_{u(n)} \right\| \tag{12.3}$$

It follows that the least-squares weight vector $w_{LS}(n)$, must satisfy the equation:

$$R(n)w_{LS}(n) + u(n) = 0 \tag{12.4}$$

Since $R(n)$ is an upper triangular matrix the weights can be solved using back-substitution. QR decomposition is an extension of this QR factorization, which enables the matrix to be triangularized again when new data enter the data matrix, without having to compute the triangularization from the

original square matrix format. In other words, it updates the old triangular matrix when new data are entered. The data matrix $X(n)$ and the measurement vector $y(n)$ at time n can be represented in a recursive manner by the previous resulting matrix and vector and the new data, such that:

$$X(n) = \begin{bmatrix} \lambda(n)X(n-1) \\ \underline{x}^T(n) \end{bmatrix} \tag{12.5}$$

and

$$y(n) = \begin{bmatrix} \lambda(n)y(n-1) \\ \underline{y}(n) \end{bmatrix} \tag{12.6}$$

where $\underline{x}^{\mathrm{T}}(n)$ and $\underline{y}(n)$ form the appended row at time n. A square root form of the algorithm is achieved as follows:

$$Q^T \begin{bmatrix} \lambda^{0.5}R(n-1) \\ \underline{x}^T(n) \end{bmatrix} W_{LS}(n) = Q^{\mathrm{T}}(n) \begin{bmatrix} \lambda^{0.5}u(n-1) \\ \underline{y}(n) \end{bmatrix} + Q^{\mathrm{T}}(n)e(n) \tag{12.7}$$

where $\beta = \lambda^{0.5}$. This then gives:

$$Q^T \begin{bmatrix} \beta(n)R(n-1) & \beta(n)u(n-1) \\ \underline{x}^T(n) & \underline{y}(n) \end{bmatrix} = \begin{bmatrix} R(n) & u(n) \\ O & \alpha(n) \end{bmatrix} \tag{12.8}$$

This is computed to give:

$$\begin{bmatrix} R(n) \\ O \end{bmatrix} W_{LS}(n) = \begin{bmatrix} u(n) \\ \alpha(n) \end{bmatrix} \tag{12.9}$$

$\alpha(n)$ is related to the a posteriori least-squares residual, $e(n)$, at time n such that:

$$e(n) = \alpha(n)\gamma(n) \tag{12.10}$$

where $\gamma(n)$ is the product of cosines generated in the course of eliminating $\underline{x}^{\mathrm{T}}(n)$.

The high-level dependence graph realization of the QR solution for RLS is shown in Figure 12.5.

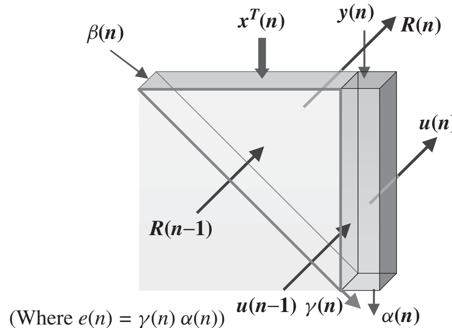


Figure 12.5 High-level dependence graph for the QR-RLS solution

12.4.4 Givens Rotations Used for QR Factorization

A family of numerically stable and robust RLS algorithms has evolved from a range of QR decomposition methods such as Givens rotations (Gentleman and Kung 1981, Givens 1958, McWhirter 1983), CORDIC (Hamill 1995), and Householder transformations (Cioffi 1990, Liu *et al.* 1990, 1992, Rader and Steinhardt 1986). Givens rotations are orthogonal plane rotations, used to eliminate elements within a matrix. By applying a series of successive Givens rotations, a matrix can be triangularized by eliminating the elements beneath the diagonal. This operation is known as QR factorization whereby a matrix $X(n)$ is decomposed into an upper triangular matrix $R(n)$ and an orthogonal matrix $Q(n)$, such that:

$$X(n) = Q(n)R(n) \tag{12.11}$$

The $X(n)$ matrix is pre-multiplied by rotation matrices one element at a time. The rotation parameters are calculated so that the sub-diagonal elements of the first column are zeroed. Then the next column's sub-diagonal elements are zeroed and so forth, until an equivalent upper triangular matrix is formed.

Givens performs this operation through a sequence of rotations, which are best described by a trivial example using a 2×3 matrix, as shown below:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{11} & a_{12} & a_{13} \end{bmatrix} \tag{12.12}$$

This matrix is transformed into a pseudo-triangular matrix by eliminating the element, a_{21}. This is achieved by multiplying the matrix through by the rotation matrix:

$$\begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix}$$

Thus:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{11} & a_{12} & a_{13} \end{bmatrix} \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix} =$$

$$\begin{bmatrix} a_{11}\cos\alpha + a_{21}\sin\alpha & a_{12}\cos\alpha + a_{22}\sin\alpha & a_{13}\cos\alpha + a_{23}\sin\alpha \\ -a_{11}\sin\alpha + a_{21}\cos\alpha & -a_{12}\sin\alpha + a_{22}\cos & \alpha - a_{13}\sin\alpha + a_{23}\cos\alpha \end{bmatrix}$$

To eliminate a_{21} we need to solve for $[-a_{11}\sin\alpha + a_{21}\cos\alpha] = 0$

Therefore, from trigonometry:

$$\sin\alpha = a_{21}/\sqrt{a_{11}^2 + a_{21}^2}$$

$$\cos\alpha = a_{11}/\sqrt{a_{11}^2 + a_{21}^2}$$

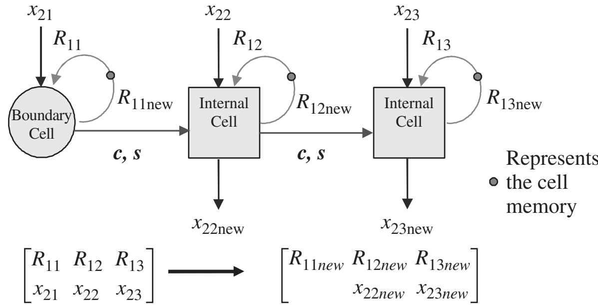


Figure 12.6 Givens rotations

Applying the rotation to eliminate a_{21} results in the pseudo-triangular matrix:

$$\begin{bmatrix} a_{11\ new} & a_{12\ new} & a_{13\ new} \\ & a_{12\ new} & a_{13\ new} \end{bmatrix} \tag{12.13}$$

This function lends itself for implementation on a triangular systolic array, as shown in Figure 12.6, consisting of two types of cell, referred to as a boundary cell (BC) and an internal cell (IC). The elements on which the rotations are performed are x and R, where x is the input value into the cell and R is the value that is held in the memory of that cell. The rotation parameters, cos α, and sin α are calculated in the BC (denoted by a circle) so that the input x value to that cell is eliminated, and the R value within that cell is updated according to that rotation and stored for the next iteration. The rotation parameters are then passed along the entire row unchanged through the ICs (denoted by a square) continuing the rotation. The dependence graph in Figure 12.6 represents the elimination of x_{21}, relating to the sub-diagonal element, a_{21} in the first column shown in the previous example. In effect, the R and x values are considered as a polar coordinate, (R, x).

Eliminating the x input to the BC is achieved by rotating R through an angle α, such that:

$$R_{new} = R\cos\alpha + x\sin\alpha = \frac{R^2 + x^2}{\sqrt{R^2 + x^2}} = \sqrt{R^2 + x^2}$$

where

$$c = \cos\alpha = \frac{R}{R_{new}}$$

and

$$s = \sin\alpha = \frac{x}{R_{new}}$$

The same rotation within the BC is carried on throughout the ICs:

$$R_{new} = cR + sx$$

$$x_{new} = cx - sR$$

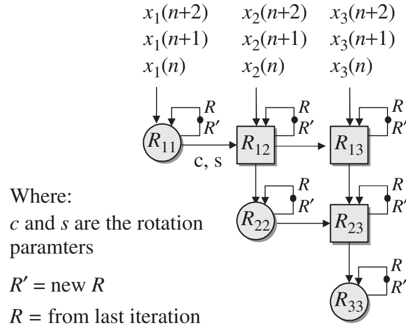


Figure 12.7 QR factorization applied to a 3×3 matrix

By concatenating successive Givens rotations QR factorization can be implemented for an $n \times n$ matrix. The diagram Figure 12.7, shows an example of a 3×3 matrix. It has been pipelined allowing the R values to be fed back into the cells for the next iteration.

The following section gives a more detailed look at the process of developing a hardware architecture from a mathematical algorithm for the RLS algorithm solved by QR decomposition using Givens rotations.

12.5 Algorithm to Architecture

In the previous section, details have been given on the analysis of adaptive algorithms and in particular RLS solved by QR decomposition using Givens rotations. A key aspect in achieving a high-performance circuit implementation is to ensure an efficient mapping of the algorithm onto silicon hardware. This may involve developing a hardware architecture in which independent operations are performed in parallel so as to increase the throughput rate. In addition, pipelining may be employed within the processor blocks to achieve faster throughput rates. One architecture that uses both parallelism and pipelining is a systolic array. Its processing power comes from the concurrent use of many simple cells rather than the sequential use of a few very powerful cells. The result is a regular array of identical processors with only local interconnections. As technology advances, gate delay is no longer the dominating factor controlling the performance of circuits. Instead, interconnection lengths have the most influence over the critical paths and power consumption of a circuit, hence the importance of keeping connections local.

Figure 12.8 illustrates the process from algorithm to architecture with the starting point within the diagram being the RLS algorithm solved by QR decomposition. The next stage of the diagram depicts the RLS algorithm solved through QR decomposition using a sequential algorithm. That is, for each iteration of the algorithm, a new set of values are input to the equations thus continuously progressing toward a solution. The QR operation can be depicted as a triangular array of operations. The data matrix is input at the top of the triangle and with each row another term is eliminated so to eventually result in an upper triangular matrix. The dependence graph (DG) in Figure 12.8 depicts this triangularization process. The cascaded triangular arrays within the diagram represent the iterations through time, i.e. each one represents a new iteration. The arrows between the cascaded arrays highlight the dependency through time.

From the DG, a suitable SFG can be derived and from this representation, an architecture can be developed. The following sections give a brief overview of each of these stages as depicted in Figure 12.8.

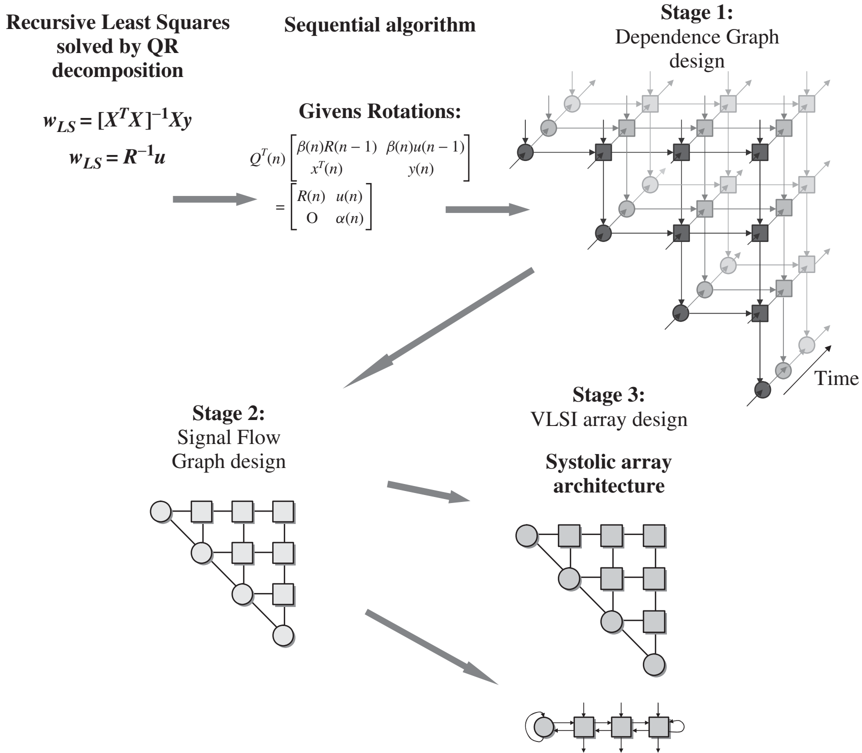


Figure 12.8 From algorithm to architecture

12.5.1 Dependence Graph

The dependences between data can be identified in a DG, allowing the maximum level of concurrency to be identified by breaking the algorithm into nodes and arrows. The nodes outline the computations and the direction of the arrows shows the dependence of the operations. This is shown for the QR algorithm by the three-dimensional DG in Figure 12.9. The diagram shows three successive QR iterations, with 'dependence arcs' connecting the dependent operations. Some of the variable labels have been omitted for clarity.

The new data are represented by $\underline{x}^{T}(n)$. The term n represents the iteration of the algorithm. In summary, the QR array performs the rotation of the input $\underline{x}^{T}(n)$ vector with R values held within the memory of the QR cells so that each input x value into the BCs, are rotated to zero. The same rotation is continued along the line of ICs via the horizontal arrows between QR cells. From this DG, it is possible to derive a number of SFG representations. The most obvious projection which is used here, is to project the DG along the time (i.e. R) arrows.

12.5.2 Signal Flow Graph

The transition from the DG to SFG is clearly depicted in Figure 12.10. To derive the SFG from the DG, the nodes of the DG are assigned to processors, then their operations are scheduled on these processors. One common technique for processor assignment is linear projection of all identical nodes along one straight line onto a single processor. This is represented mathematically by the

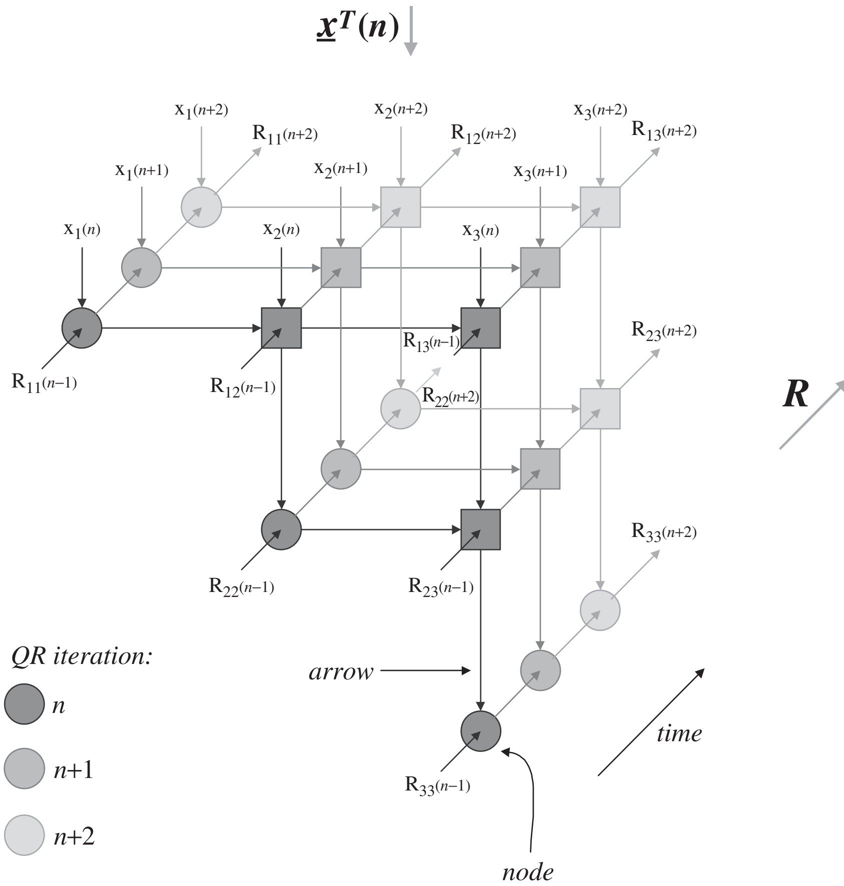


Figure 12.9 Dependence graph for QR decomposition

projection vector $\boldsymbol{d}$ (Figure 12.10). Linear scheduling is then used to determine the order in which the operations are performed on the processors. The schedule lines in Figure 12.10 indicate the operations that are performed in parallel at each cycle. Mathematically they are represented by a schedule vector $\boldsymbol{s}$ normal to the schedule lines, which points in the direction of dependence of the operations, i.e. it shows the order in which each line of operations is performed.

There are two basic rules that govern the projection and scheduling, and ensure that sequence of operations is retained. Given a DG and a projection vector $\boldsymbol{d}$, the schedule is permissible if and only if:

- all the dependence arcs flow in the same direction across the schedule lines;
- the schedule lines are not parallel with the projection vector $\boldsymbol{d}$.

In the QR example (Figure 12.10), each triangular array of cells within the DG represents one QR update. When cascaded, the DG represents a sequence of QR updates. By projecting along the time axis, all the QR updates may be assigned onto a triangular SFG, as depicted in the Figure 12.10(b).

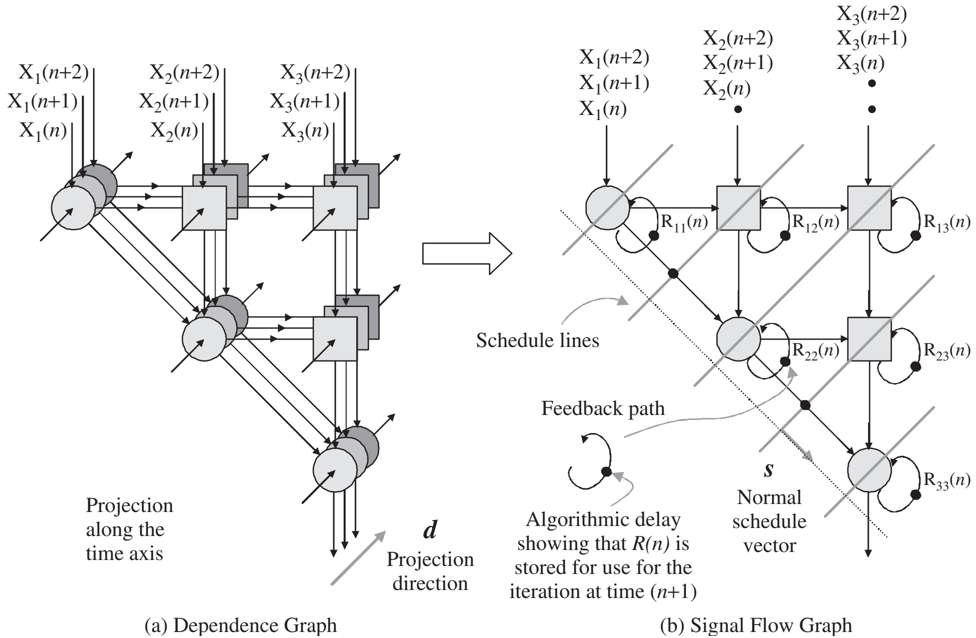


Figure 12.10 From dependence graph to signal flow graph

In the DG, the R values are passed through time from one QR update to another, represented by the cascaded triangular arrays. This transition is more concisely represented by the loops in Figure 12.10(b), that feed the R values back into the cells via an algorithmic delay needed to hold the values for use in the next QR update. This is referred to as a recursive loop.

The simplicity of the SFG is that it assumes that all operations performed within the nodes take one cycle, as with the algorithmic delays, represented by small black nodes. These algorithmic delays partition the iterations of the algorithm and are a necessary part of the algorithm. The result of the SFG is a more concise representation of the algorithm than the DG.

The remainder of the chapter gives a detailed account of the processes involved in deriving an efficient architecture and hence hardware implementation of the SFG representation of the algorithm. In particular, emphasis is on creating an intuitive design that will be parameterizable, therefore enabling a fast development for future implementations.

12.5.3 Systolic Implementation of Givens Rotations

The resulting systolic array for the conventional Givens RLS algorithm is shown in Figure 12.11. Note that the original version, proposed by Gentleman and Kung (1981), shown in Figure 12.7, did not include the product of cosines formed down the diagonal line of BCs, and is represented by the arrows connecting the BCs down the diagonal. This modification was made by McWhirter (1983) and is of significant importance as it allows the QR array to perform both the functions for calculating the weights and also enables it to operate as the filter itself, that is, the error residual (*a posteriori* error) may be found without the need for weight vector extraction. This offers an attractive solution in applications such as adaptive beamforming, where the output of interest is the error residual. The definitions for the BC and ICs are depicted in Figures 12.12 and 12.13 respectively.

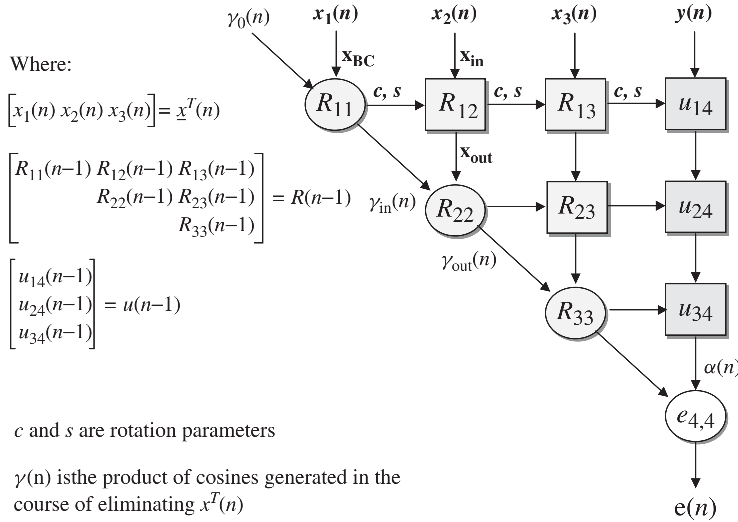


Figure 12.11 Systolic QR array for the RLS algorithm

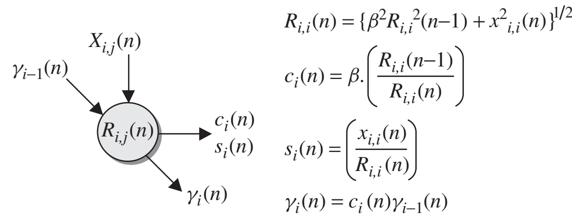


Figure 12.12 BC for QR-RLS algorithm

The data vector $\underline{x}^{\mathrm{T}}(n)$ is input from the top of the array and is progressively eliminated, by rotating it within each row of the stored triangular matrix $R(n-1)$ in turn. The rotation parameters c and s are calculated within a BC such that they eliminate the input, $x_{i,i}(n)$. These parameters are then passed unchanged along the row of ICs, continuing the rotation. The output values of the ICs, $x_{i+1,j}(n)$ become the input values for the next row. Meanwhile, new inputs are fed into the top of the array, and so the process repeats. In the process, the $R(n)$ and $u(n)$ values are updated to account for the rotation and then stored within the array to be used on the next cycle.

For the RLS algorithm, the implementation of the forget factor λ, and the product of cosines γ, need to be included within the equations. Therefore the operations of the BC and ICs have been modified accordingly. A notation has been assigned to the variables within the array. Each R and u term has a subscript, denoted by (i, j), which represents the location of the elements within the R matrix and u vector. A similar notation is assigned to the X input and output variables. The cell descriptions for the updated BC and ICs are shown in Figures 12.12 and 12.13 respectively. The subscripts are coordinates relating to the position of the cell within the QR array.

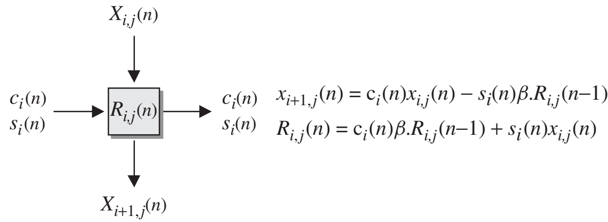


Figure 12.13 IC for QR-RLS algorithm

12.5.4 Squared Givens Rotations

There are division and square root operations within the BC cell computation for the standard Givens rotations (Figure 12.12). There has an extensive body of research into deriving Givens rotation QR algorithms which avoid these complex operations, while reducing the overall number of computations (Cioffi and Kailath 1984, Döhler 1991, Hsieh *et al*. 1993, Walke 1997). One possible QR algorithm is the squared Givens rotation (SGR (Döhler 1991)). Here the Givens algorithm has been manipulated to remove the need for the square root operation within the BC and half the number of multipliers in the ICs. Studies by Walke (1997) showed that this algorithm provided excellent performance within adaptive beamforming at reasonable wordlengths (even with mantissa wordlengths as short as 12 bits with an increase of 4 bits within the recursive loops). This algorithm proves to be a suitable choice for the adaptive beamforming design. Figure 12.14 depicts the SFG for the SGR algorithm, and includes the BC and IC descriptions.

This algorithm still requires the dynamic range of floating-point arithmetic, but offers reduced size over fixed-point algorithms, due to the reduced wordlength and operations requirement. It has the added advantage of allowing the use of a multiply–accumulate operation to update R. The simplicity of the recursive loop is a key advantage as the number of clock cycles within this loop will govern the maximum throughput for a particular clock frequency. For example, if a QR array had 10 clock cycles within the recursive loop then there would need to be 10 clock cycles between successive QR iterations, to enable the value $R(n)$ to be calculated in time for use in iteration $n + 1$.

At little cost in hardware, the wordlength of the accumulator can be increased to improve the accuracy to which R is accumulated, while allowing the overall wordlength to be reduced. This has been referred to as the enhanced SGR algorithm, (E-SGR) (Walke 1997).

However, even with the level of computation reduction achievable by the SGR algorithm, the complexity of the QR cells is still large. In addition the number of processors within the QR array increases quadratically with the number of inputs, such that for an N-input system, $(N^2 + N)/2$ QR processors are required; furthermore, implementing a processor for each cell could offer data rates far greater than those required by most applications. The following section details the process of deriving an efficient architecture with generic properties for implementing the SGR QR-RLS algorithm.

12.6 Efficient Architecture Design

With the complexity of the SGR QR-RLS algorithm coupled by the number of processors increasing quadratically with the number of inputs, it is vital to generate efficient QR array architectures tailored to the applications that meet desired performance with the lowest area cost. Consider an example

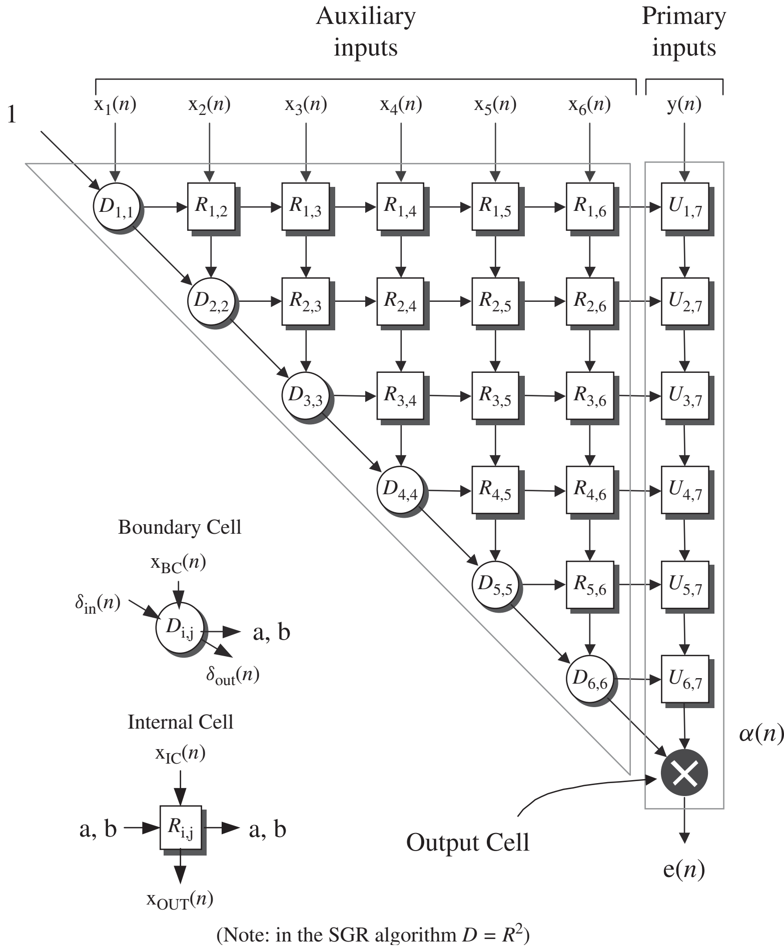


Figure 12.14 Squared Givens rotations QR-RLS algorithm

application consisting of 40 inputs, but only requiring a throughput of 1 MSPS. Implementing the full QR array, using a clock rate of 200 MHz and a recursive loop delay of 4 clock cycles (a value which is used throughout the duration of this chapter). This means that the new inputs could be fed into the QR array every 4 clock cycles, therefore providing throughput capability of 200 MHz/4 = 50 MSPS. Quite clearly, this performance is not required and the design could benefit from a reduction in hardware.

This is achievable by mapping the triangular functionality down onto a smaller array of processors. The triangular shape of the QR array, in addition to the position of the BC operations along the diagonal, complicates the process of deriving an efficient architecture. Figure 12.15 shows an example of a simple mapping of the QR cells onto a linear architecture by a projection from left to right onto N processors. There are two issues with such a mapping. First, both BC and IC operations are mapped onto the same processor; from Figure 12.14 it can be seen that there are distinct differences between these operations. Second, the processors of the mapped architecture are

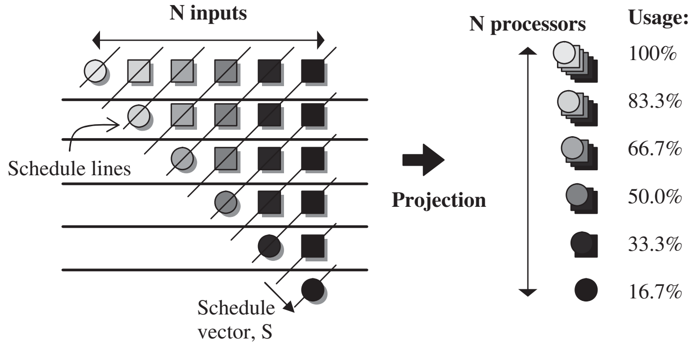


Figure 12.15 Simple linear array mapping

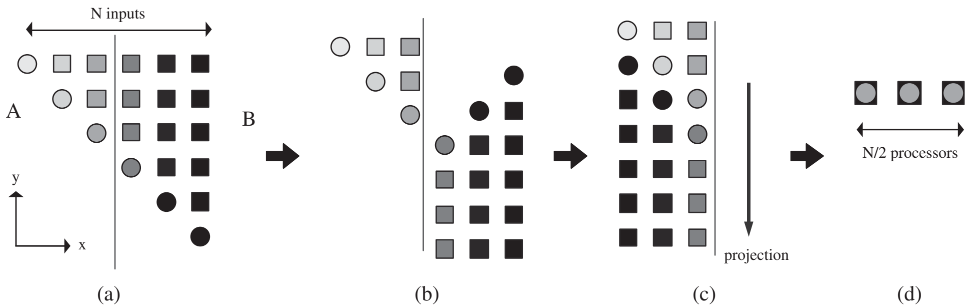


Figure 12.16 Radar mapping (Rader 1992, 1996)

not used efficiently, with only the first being exploited to its full capacity. This efficiency reduces down the column of processors, leading to an overall efficiency in the region of 60%–certainly not an optimum use of resources.

Rader (1992, 1996) produced an efficient architecture by manipulation of the triangular shape before assigning the operations to the processors, as depicted in Figure 12.16. Part B of the QR array is mirrored in the x-axis and then folded back onto the rest of the array. This results in a rectangular array of cells which can then be mapped down onto a linear architecture consisting of $N/2$ processors. However, the processors still need to perform both QR operations. One option is the use of the CORDIC algorithm (Hamill 1995) which has strong archiecture similarities between BC and IC, another could be to design a generic QR cell (Lightbody *et al*. 2007) based on core arithmetic blocks on which to build up cell functionality. Other mappings for the QR array also exist, one solution (Tamer and Ozkurt 2007) used a tile structure on which to map the QR cells, leading to processors performing only the IC operation and also ones performing both functions.

Another mapping (Walke 1997), manages to maintain an efficient architecture while keeping the BC and IC operations to distinct processors. This is achieved by cleverly manipulating the triangular shape of the array so as to align all the BC operations onto one column of a rectangular array of processors, while all the IC operations are mapped to the other columns. It does this by folding and rotating parts of the QR array, as depicted in Figure 12.17 for an example 7-input triangular array. The resulting mapping assigns the triangular array of $2m^2 + 3m + 1$ cells (i.e. $N = 2m + 1$

inputs) onto a linear architecture, with local interconnections, comprising of 1 BC processor and m IC processors, all used with 100% efficiency. The method is described in greater detail elsewhere (Lightbody 1999, Lightbody *et al*. 2003, Walke 1997).

For clarity, each QR operation is assigned a co-ordinate originating from the R (or U) term calculated by that operation, i.e. the operation $R_{1,2}$ is denoted by the coordinate, $1, 2$, and $U_{1,7}$ is denoted by $1, 7$. To simplify the explanation, the multiplier at the bottom of the array is treated as a BC, denoted by $7, 7$.

The initial aim of this mapping is to manoeuvre the cells so that they form a locally interconnected regular rectangular array. This can then be partitioned evenly into sections, each to be assigned to an individual processor. This should be done in such a way to achieve 100% cell usage and a nearest neighbour connected array. Obtaining the rectangular array is achieved through the following four stages. The initial triangular array is divided into two smaller triangles A and B. A cut is then made after the $(m + 1)^{\text{th}}$ BC at right angles to the diagonal line of BCs (Figure 12.17a). Triangle A forms the bottom part of a rectangular array, with $m + 1$ columns and $m + 1$ rows.

Triangle B now needs to be manipulated so that it can form the top part of the rectangular array. This is done in two stages. By mirroring triangle B first in the x-axis, the BCs are aligned in such a way that they are parallel to the BCs in the triangle A, forming a parallelogram, as shown in Figure 12.17(b). The mirrored triangle B is then moved up along the y-axis and left along the x-axis to a position above A, forming the rectangular array (Figure 12.17(c)). As depicted, the BC operations are aligned down two columns and so the rectangular array is still not in a suitable format for assigning operations onto a linear architecture.

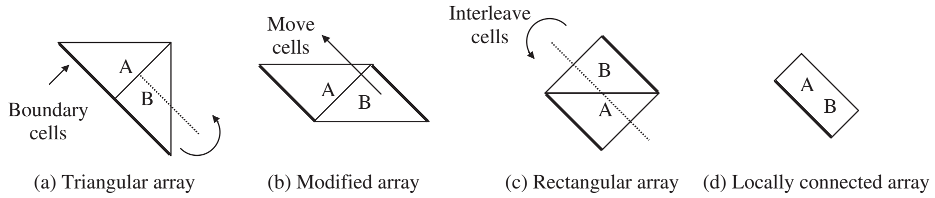
The next stage aims to fold the large rectangular array in half so that the two columns of BC operations are aligned along one column. This fold interleaves the cells so that a compact rectangular processor array (Figure 12.17(d)) is produced. From this rectangular processor array, a reduced architecture can be produced by projection down the diagonal onto a linear array, with all the BC operations assigned to one BC process and all the IC operations assigned to a row of m IC processors (Figure 12.17e). The resulting linear architecture is shown in more detail in Figure 12.18.

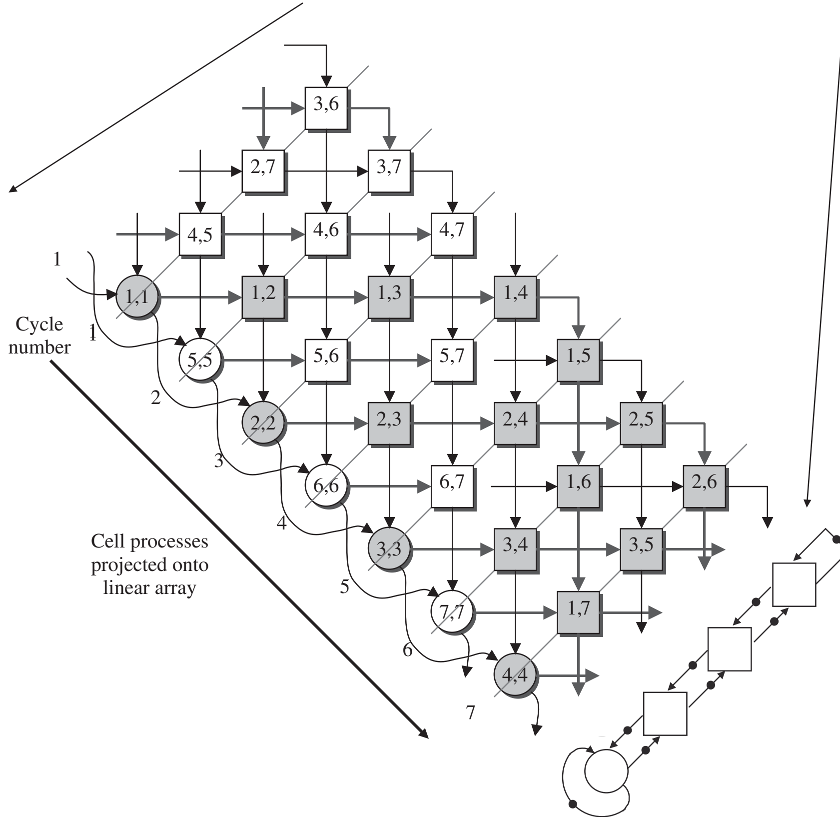
There are lines drawn through each row of processors within the rectangular array of operations in Figure 12.17(e) (labelled 1–7). These represent the operations that need to be performed on each cycle of the resulting linear array architecture. This is what is termed as the schedule of operations and is more compactly denoted by $\boldsymbol{s}$, the schedule vector, an arrow that is perpendicular to the schedule lines. At this stage of the analysis, it is assumed that each cell processor takes one clock cycle. There are registers present on the all the processor cell outputs of the resulting linear array to maintain this schedule. Multiplexers are placed on the inputs of the QR cells to control the data inputs, whether from the system inputs or from the adjacent cells. The bottom multiplexers govern the different directions of data flow that occur between rows of the original array.

The original QR array cells store the R values from one iteration to the next. This same storage needs to be performed for the reduced architecture, therefore requiring a number of R values to be stored within the recursive loops of the cells for multiple clock cycles. One solution is to hold the values locally within the recursive datapaths of the QR cells, rather than external memory, i.e. the values are pipelined locally to delay them until they are needed. Some of the required delays within the recusive loop are met by the latency of existing operations within the loop and the remainder are achieved by inserting additional registers. External memory could be suitable under some specifications.

12.6.1 Scheduling the QR Operations

The derivation of the architecture is only a part of the necessary development. A more complex task can be the determination of a valid schedule that ensures that the data required by each set of operations is available at the time of execution, while maintaining efficiency. This implies that the





(e) Projection of cells on to a linear array of locally connected processors

Figure 12.17 Interleaved processor array. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

data must flow across the schedule lines in the direction of the schedule vector. The rectangular processor array in Figure 12.17(e) contains all the operations required by the QR algorithm, showing the sequence that they are to be implemented on the linear architecture. Therefore, this diagram can be used to show the schedule of the operations to be performed on the linear architecture. An analysis of the scheduling and timing issues can now be refined. Looking at the first schedule line, it can be seen that operations from two different QR updates have been interleaved. The shaded cells

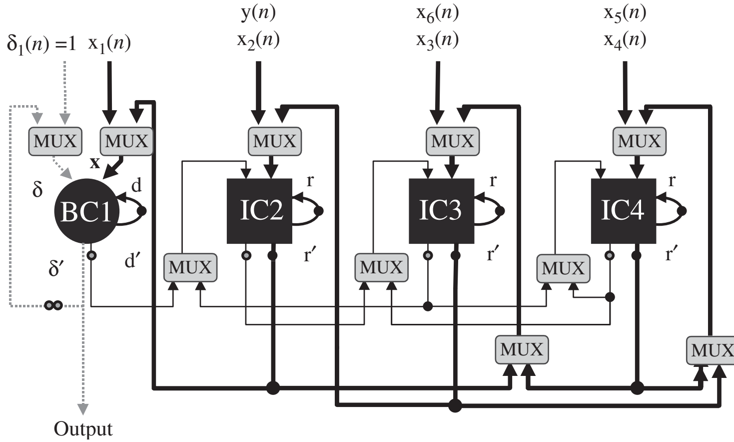


Figure 12.18 Linear architecture for a 7-input QR array. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

represent the current QR update at time n and the unshaded cells represent the previous unfinished update at time $n - 1$. Effectively the QR updates have been interleaved. This is shown in more clarity in Figure 12.19. The first QR operation begins at cycle =1 then after $2m + 1$ cycles of the linear architecture, the next QR operation begins. Likewise, after a further $2m + 1$ cycles the third QR operation is started. In total, it takes $4m + 1$ cycles of the linear architecture to complete one specific QR update.

The QR cells need to be able to take the x inputs from external system inputs, i.e. from the snapshots of data forming the input $x(n)$ matrix and $y(n)$ vector, as depicted in Figure 12.19. The external inputs are fed into the linear architecture every $2m + 1$ clock cycles. They will also take inputs, which are internal to the linear array. The mapping process has enabled these interconnections to be kept local which is a major benefit.

If each QR cell takes a single clock cycle to produce an output then there will be no violation of the schedule shown in Figure 12.17. However, additional timing issues must be taken into account as processing units in each QR cell have detailed timing requirements. The retiming of the operations is discussed in more detail later in Section 12.8.

Note that the processor array highlighted in Figure 12.19 is equivalent to the processor array given in Figure 12.17(e). This processor array is the key starting point from which to develop a generic QR architecture.

12.7 Generic QR Architecture

The technique presented so far has been applied to a QR array with only one primary input, that is, one y input. To develop a generic QR architecture the number if primary inputs would need to be variable. This would result in a QR array consisting of a triangular part and a rectangular part (Figure 12.20), the sizes of which are determined by the number of auxiliary and primary inputs, respectively. Typically, the number of inputs to the triangular part is at least a factor greater than the number of inputs to the rectangular part, with example numbers for the radar application being 40 inputs for the triangular part and in the region of only 2 for the rectangular part.

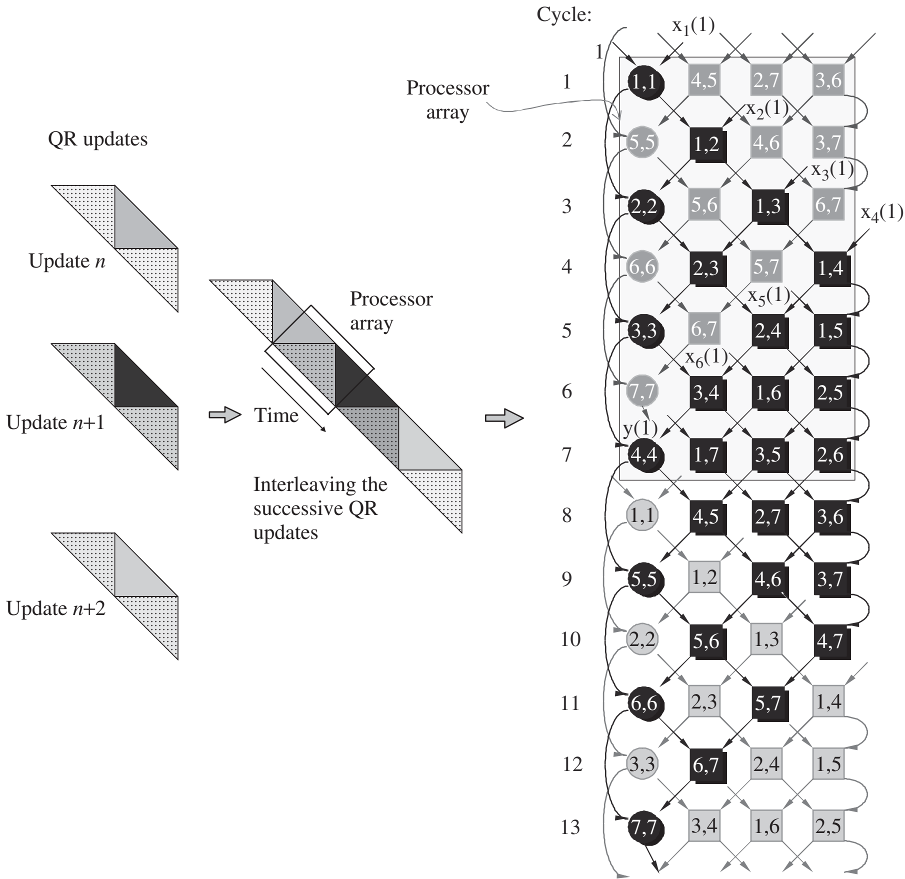


Figure 12.19 Interleaving successive QR operations. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

This section extends the mapping technique already presented to implement architectures for the generic QR array consisting of both the triangular and rectangular parts. Different levels of hardware mapping are applied, providing a range of suitable architectures based on the original linear array. The number of IC processors may be reduced further, or multiple linear arrays may be combined, depending on the performance requirements for the application. Note that the connections have been removed from Figure 12.20 and in following diagrams, in order to reduce the complexity of the diagram and aid clarity.

12.7.1 Processor Array

In the previous section, the triangular structure of the QR array was manipulated into a rectangular processor array of locally interconnected processors, as shown in Figure 12.17(d). From this starting point the operations can be mapped onto a reduced architecture. A simplified method for creating the processor array is demonstrated in the following example.

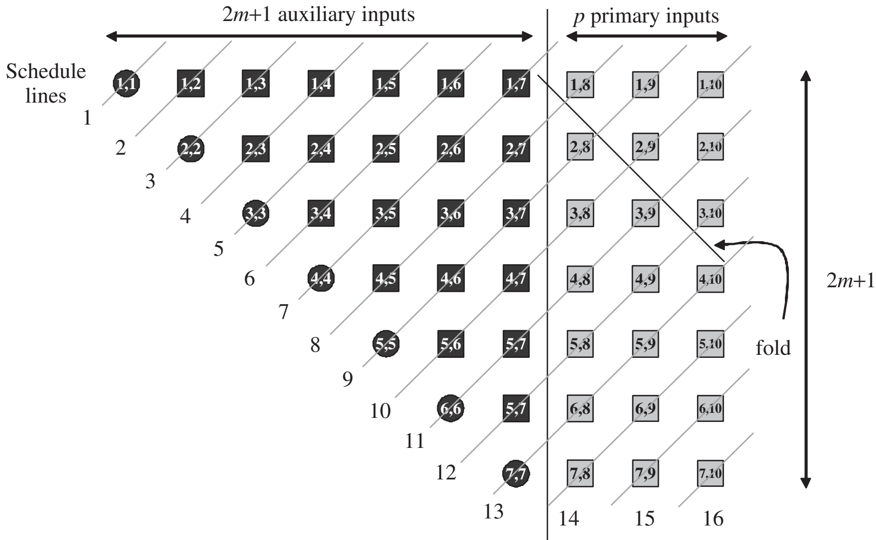


Figure 12.20 Generic QR array with $(2m + 1)$ auxiliary inputs and p primary inputs. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

The processor array is obtained through two steps. First, a fold is made by folding over the corner of the array after the mth cell from the right-hand side, as depicted in Figure 12.20. The cells to be folded from the corner of the array are then interleaved between the rows of unfolded cells, as shown in Figure 12.21. For the next stage, successive QR updates need to be considered. The gaps within the structure in Figure 12.21 are removed by interleaving this QR update with the previous iteration and the next iteration. This is effectively the same process as the derivation for the linear array, as shown in Figure 12.19 for the original triangular array without the rectangular part.

The choice of position of the fold and the size of the triangular part of the array are important. By placing the fold after the mth cell from the right-hand side, a regular rectangular array of operations can be produced. This can be shown in greater detail, for the generic QR array of Figure 12.22. Just as with the triangular array, the same process applied to the triangular and rectangular part leads to a section which repeats over time and contains each of all the required QR operations. This section is referred to as the processor array. It is more clearly depicted in Figure 12.23, which shows just the repetitive section from Figure 12.22.

In the example given in Figure 12.23 the processor array contains QR operations built up from three successive QR updates, represented by the differently shaded cells. The interconnections have been maintained within this figure, highlighting the local interconnectivity of cells. The size of the processor array is determined by the original size of the triangular QR array, that is, the number of auxiliary and primary inputs; $(2m + 1)$ and p respectively. The resulting processor array has the dimensions $(2m + 1)$ rows by $(m + p + 1)$ columns, the product of which gives the total number of cells in the original array. From this processor array, a range of architectures with a variable level of hardware reduction can be obtained by dividing the array into partitions and then assigning each of the partitions to an individual processor. There are several possible variants of QR architecture, as listed next:

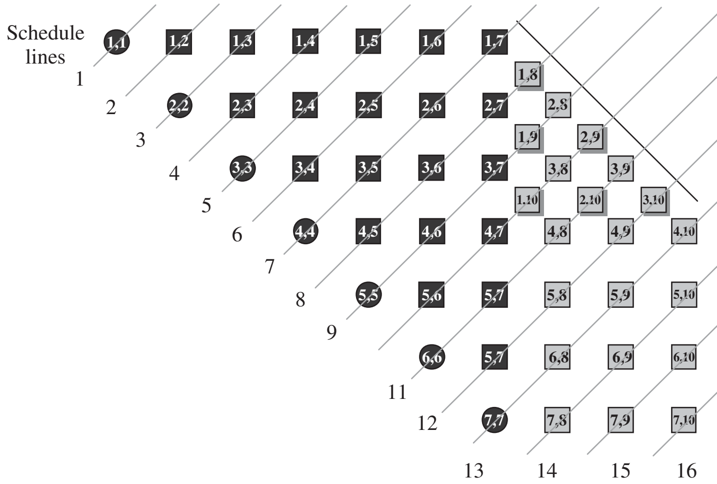


Figure 12.21 Generic QR array: Folded corner of the *m*th cell from the right-hand side. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

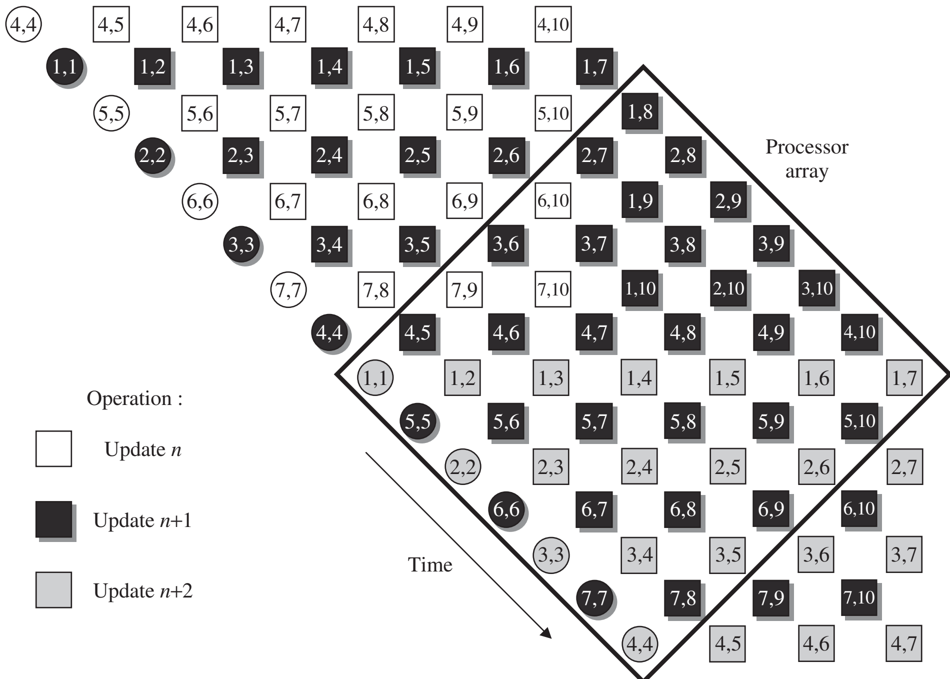


Figure 12.22 Repetitive section. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

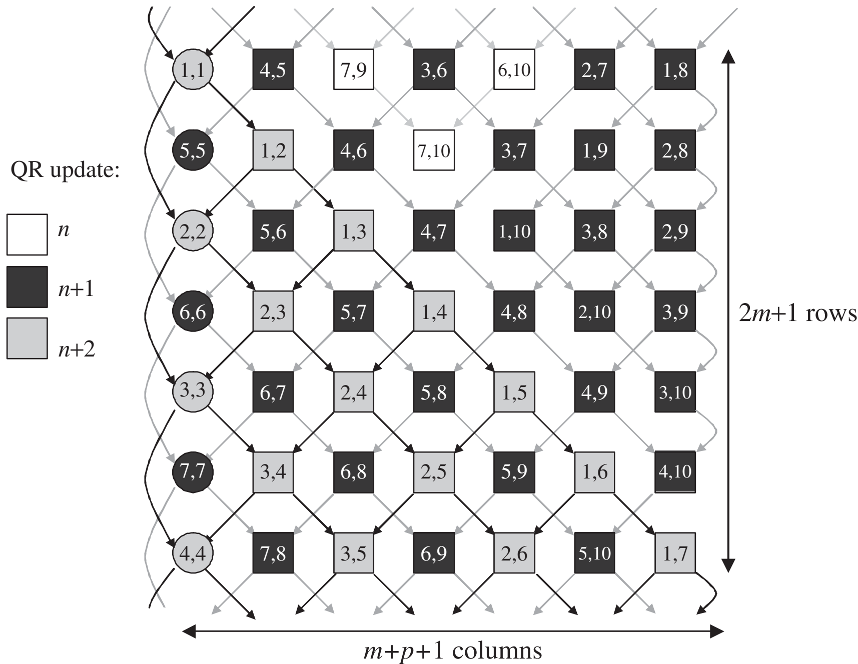


Figure 12.23 Processor array. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

Linear architecture: the rectangular array is projected down onto a linear architecture with one BC and $(m + p)$ ICs.

Rectangular architecture: the rectangular array is projected down onto a number of linear rows of cells. The architecture will have r rows, (where $1 < r = 2m + 1$), and each row will have one BC and $(m + p)$ ICs.

Sparse linear architecture: the rectangular array is projected down onto a linear architecture with one BC and less than $(m + p)$ ICs.

Sparse rectangular architecture: the rectangular array is projected down onto a number of linear rows of cells. The architecture will have r rows, (where $1 < r = 2m + 1$), and each row will have one BC and less than $(m + p)$ ICs.

The above extract is taken from Lightbody *et al.*, © 2003 IEEE

Using the same QR processor array example from Figure 12.23, examples of each type of reduced architecture are given in the following sections.

Linear Array

The linear array is derived by assigning each column of operations of the processor array onto an individual processor, resulting in a linear architecture of $m + p + 1$ processors, as shown in Figure 12.24. In total it takes 16 (i.e. $4m + p + 1$) cycles of the linear array to complete each QR

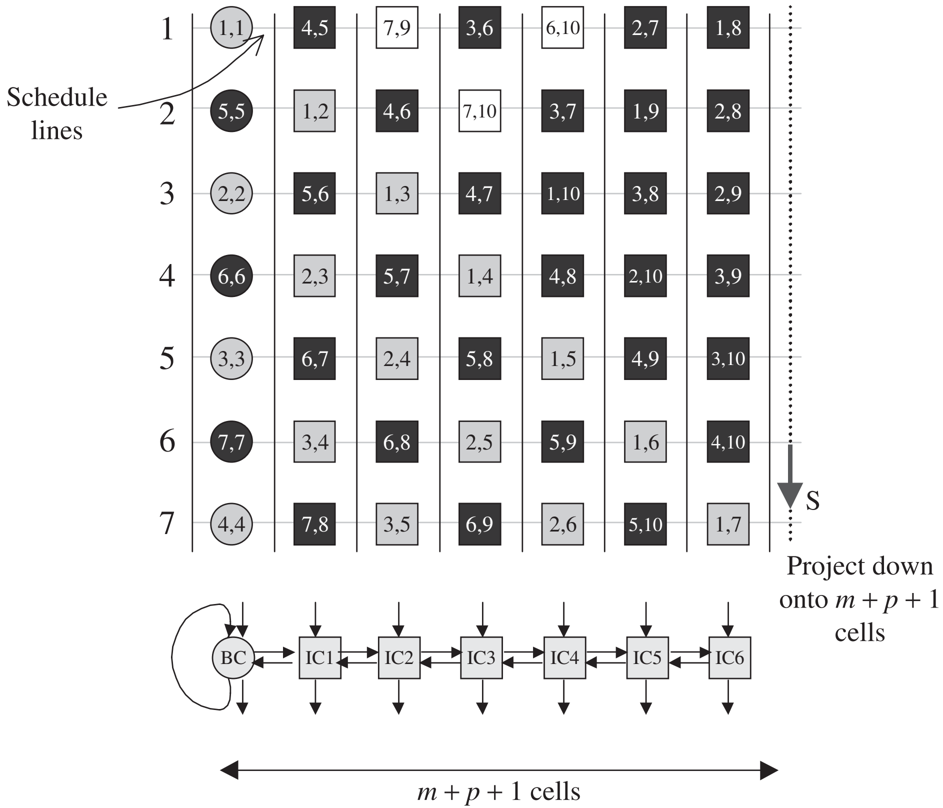


Figure 12.24 Linear array. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

operation. In addition there are 7 (i.e. $2m + 1$) cycles between the start of successive QR updates. This value is labelled as T_{QR}. Note that so far, the latency of the QR cells is considered to be one clock cycle, i.e. on each clock cycle, one row of QR operations are performed on the linear architecture. Likewise, it is also assumed that the recursive loops only have a one cycle delay. The later sections will examine the effect of multi-cycle latency, which occurs when cell processing elements with detailed timings are used in the development of the generic QR architecture.

Sparse Linear Array

A further level of hardware reduction is given in Figure 12.25, resulting in a sparse linear array. Here the number of IC processors has been halved. When multiple columns (i.e. N_{IC} columns) of IC operations are assigned to each processor then the number of iterations of the architecture is increased by this factor. Hence, for the sparse linear array, T_{QR} is expressed as the product of $2m + 1$ (used in the linear array) and N_{IC}. The schedule for the sparse linear array example is illustrated in Figure 12.26.

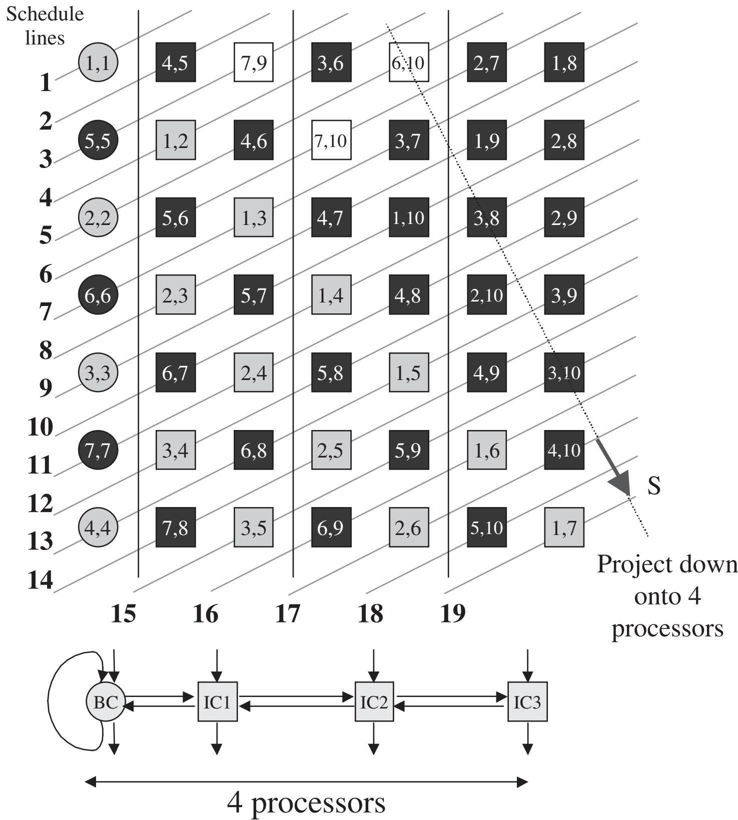


Figure 12.25 Sparse linear array. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

Rectangular Array

The processor array can be partitioned by row rather than by column so that a number of rows of QR operations are assigned to a linear array of processors. Figure 12.27 shows the processor array mapped down on an array architecture. As the processor array consisted of 7 rows, 4 are assigned to one row and 3 are assigned to the other. To balance the number of rows for each linear array, a dummy row of operations is needed and is represented by the cells marked by the letter D.

On each clock cycle the rectangular array processor executes two rows of the original processor array. Each QR iteration takes 18 cycles to be completed which is 2 more clock cycles than for the linear array due to the dummy row of operations. However, the QR updates are started more frequently. In this case T_{QR} is 4, compared with the linear array which took 7 cycles. For the array architecture, T_{QR} is determined by

$$T_{QR} = \frac{(2m+1) + N_D}{N_{rows}}$$

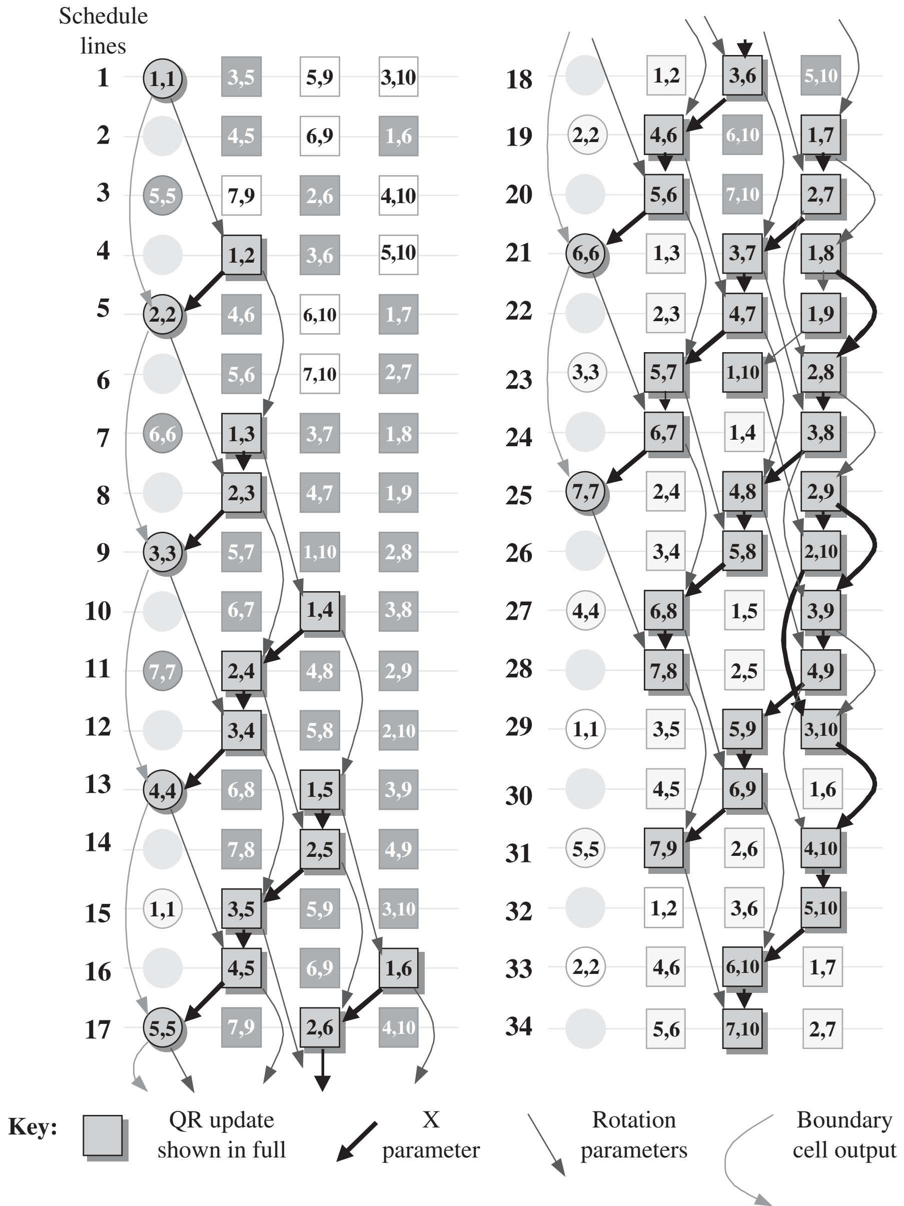


Figure 12.26 One QR update scheduled on the sparse linear array

where, N_{rows} is the number of lines of processors in the rectangular architecture, and, N_{D} is the number of rows of dummy operations needed to balance the schedule. The resulting value relates to the number of cycles of the architecture required to perform all the operations within the processor array.

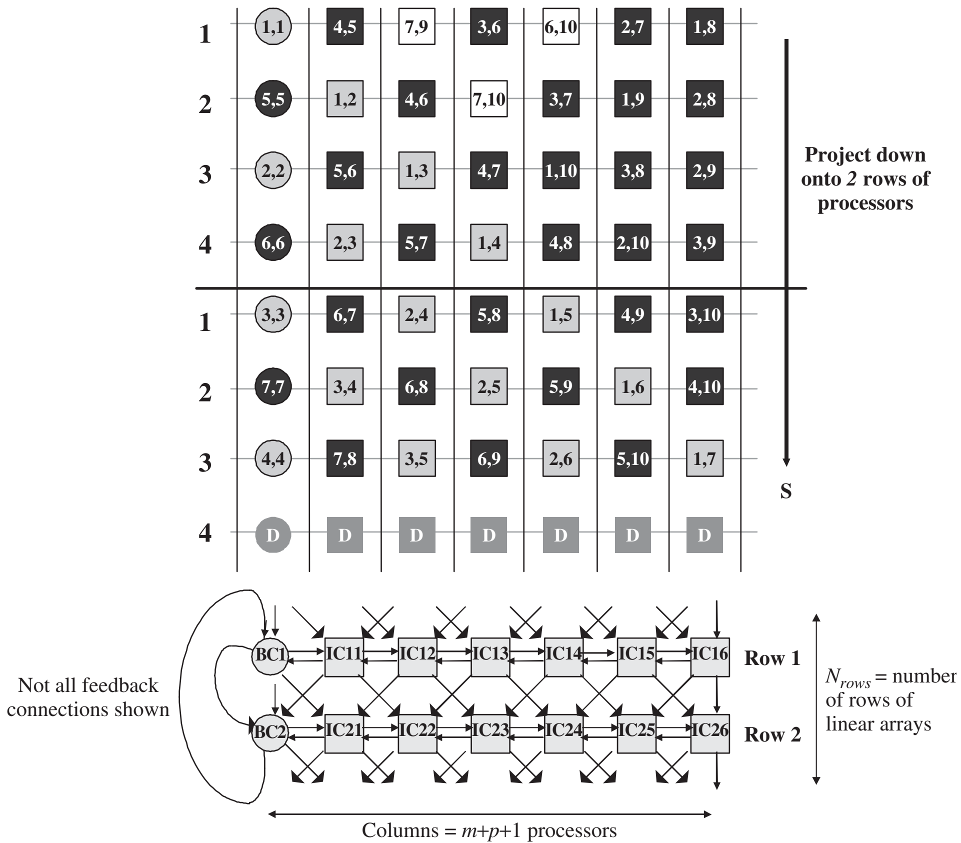


Figure 12.27 Rectangular array

Sparse Rectangular Array

The sparse rectangular array assigns the operations to multiple rows of sparse linear arrays. A number of rows of the processor array are assigned to each linear array. The columns are also partitioned so that multiple columns of operations are assigned to each IC processor, as shown in Figure 12.28.

The QR update takes 34 cycles for completion and each update starts every 7 cycles, i.e. $T_{\mathrm{QR}} = 7$. Including the term N_{IC} the equation for T_{QR} becomes:

$$T_{QR} = \frac{((2m+1) + N_D)N_{IC}}{N_{rows}}$$

For example, $T_{\mathrm{QR}} = ((2 \times 3 + 1 + 0) \times 2)/2 = 7$ cycles.

The discussion to date has concentrated on mapping QR arrays that have an odd number of auxiliary inputs. The technique can be applied to an array with an even number with a slight reduction in overall efficiency.

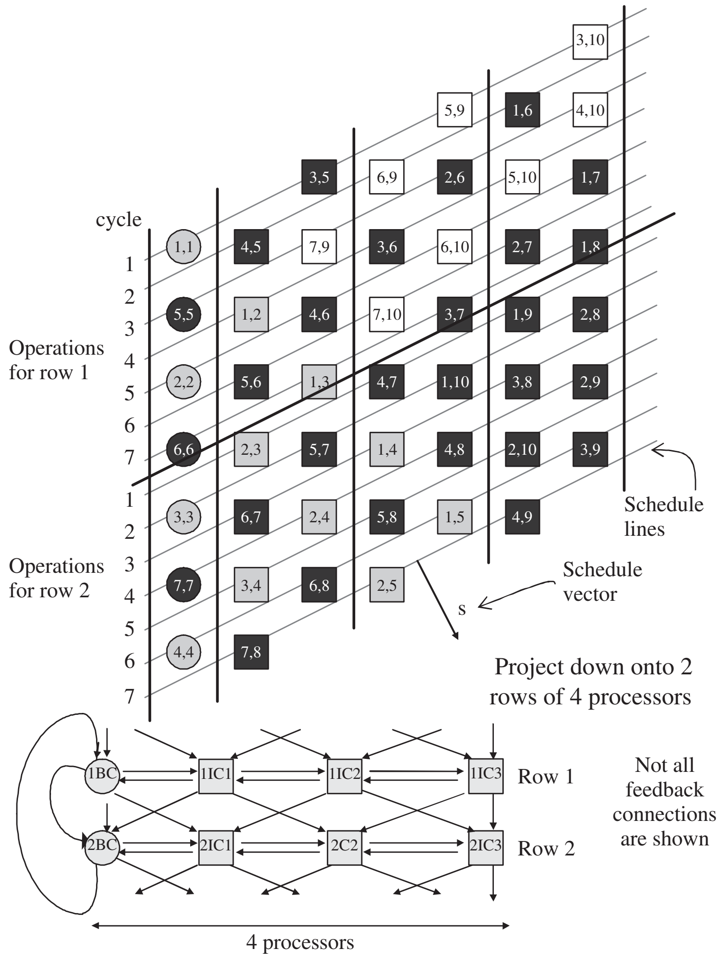


Figure 12.28 Sparse rectangular array

12.8 Retiming the Generic Architecture

The QR architectures discussed so far, have assumed that the QR cells have a latency of one clock cycle. The mapping of the architectures is based on this factor; hence there will be no conflicts of the data inputs. However, the inclusion of actual timing details within the QR cells will affect this guarantee of a valid data schedule. The arithmetic intellectual property (IP) processors (McCanny *et al*. 1997, University N 2007), used to implement the key arithmetic functions such as multiplication, addition and division, involve timing details which will impact the overall circuit timing. This was discussed in detail in Chapter 8. The overall effect of retiming is to incur variable latencies in the output datapaths of the QR cells. The effect of real timing as a result of using real hardware within the QR cells is discussed in this section. The choice for the QR array was to use floating-point arithmetic to support the dynamic range of the variables within the algorithm. The floating-point library used supported variable wordlengths and levels of pipelining, as depicted in Figure 12.29.

Floating point block	Add	SubAdd	ShftSub	Mult	Div	Round
Function	Addition: S = A + B	Adder/sub.: S = A + B when Sub = 0 else S = A − B	Shift-subtractor: $D = A - Shift(A,N)$.	Multiplier: $P = X \times Y$	Divdier: $Q = N/D$	Rounder
Symbol	A, B, S	A, B, Sub, S	A, D	X, Y, P	N, D, Q	F_{in}, R, F_{out}
Latency	0 – 3	0 – 3	0 – 1	0 – 2	0 – Mbits+1	0 – 1
Label for Latency	P_A	P_A	P_S	P_M	P_D	P_R

Figure 12.29 Arithmetic modules. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

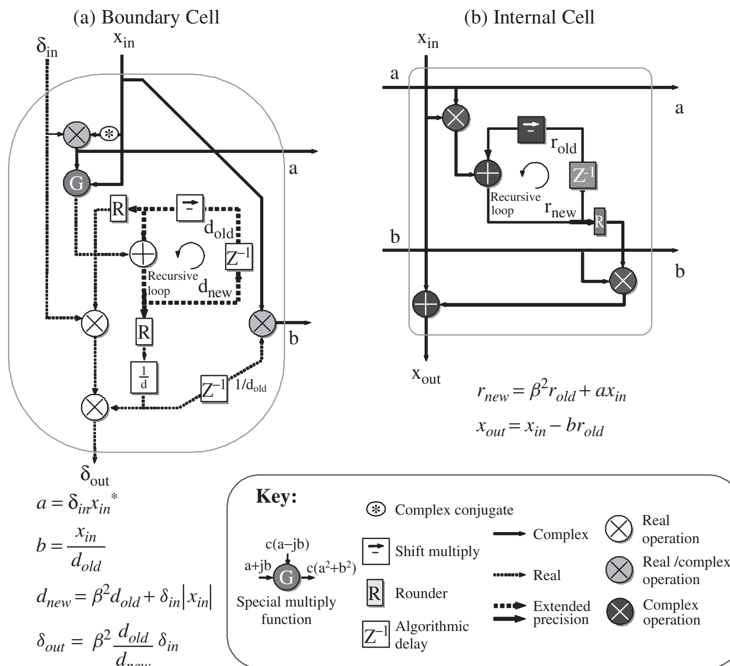


Figure 12.30 Cell SFGs for the complex arithmetic SGR QR algorithm. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

Function	Complex/Real Multiplication:	Complex Addition:	Complex Multiplication:	Special Function
Symbol	a + jb; c; ca + jcb	a + jb; c + jd; (a + c) + j(b + d)	a + jb; c + jd; (ac − bd) + j(ad + bc)	c(a − jb); a + jb; G; c(a[2] + b[2])
Real components	a jb c; P_M P_M; ca cb	a c jb jd; P_A P_A; a+c j(b + d)	a jb c jd; P_M P_M P_M P_M; P_A P_A; ac − bd j(ad + bc)	a jb ca jcb; P_M P_M; P_A; c(a[2] + b[2])
Total latency	P_M	P_A	$P_A + P_M$	$P_A + P_M$

Figure 12.31 Arithmetic modules. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

In adaptive beamforming, as with many signal-processing applications, complex arithmetic representations are needed as incoming signals contain a magnitude and a phase component. This is implemented using one signal for the real part and another for the imaginary part, and gives the BC and IC operations shown in the SFGs depicted in Figure 12.30. The floating-point complex multiplication is built up from four real multiplications and two real additions: that is, $(a + jb)(c + jd) = (ac - bd) + j(ad + bc)$. An optimization is available to implement the complex multiplication using three multiplications and five additions/subtractions. However, given that an addition is of a similar area to multiplication within floating-point arithmetic due to the costly exponent calculation, this is not beneficial. For this reason, the four-multiplication version is used. The detail of the complex arithmetic operations is given in Figure 12.31.

The SFGs for the BC and ICs are given in Figures 12.32 and 12.33, respectively. These diagrams show the interconnections of the arithmetic modules within the cell architectures. Most functions are self-explanatory except for the shift-subtracter. For small values of x, the operation $\sqrt{1-x}$ can be approximated by $1 - x^2$ which may be implemented by a series of shifts denoted by, $D = A - \text{Shift}(A, N)$. This operation is used to implement the forgetting factor, β within the feedback paths of the QR cells. This value β is close to one, therefore x is set to $(1 - \beta)$ for the function application.

There are a number of feedback loops within the QR cells, as shown in Figures 12.32 and 12.33. These store the R values from one RLS iteration to the next. These loops will be a fundamental limit to achieving a throughput rate that is close to the clock rate and, more importantly, could lead to considerable inefficiency in the circuit utilization. In other words, even when using a full QR array, the delay in calculating the new R values will limit the throughput rate.

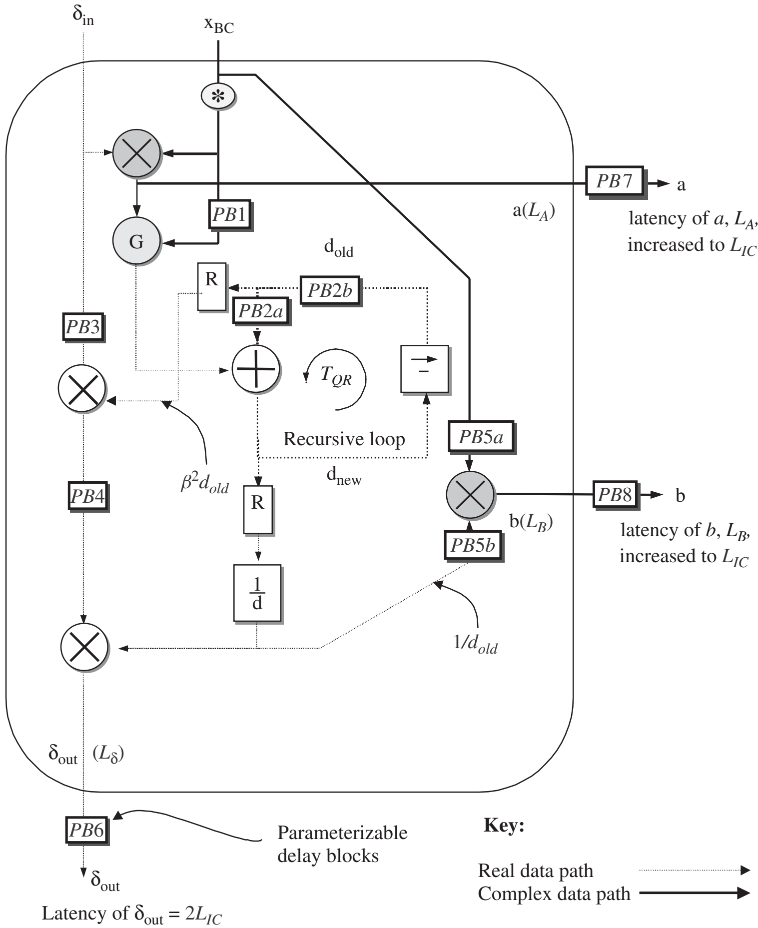


Figure 12.32 Generically retimed BC. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

Figures 12.32 and 12.33 show the QR cell descriptions with generic delays placed within the datapaths. These are there to allow for the re-synchronization of operations due to the variable latencies within the arithmetic operators, i.e. to ensure correct timing. The generic expressions for the programmable delays are listed in Tables 12.1 and 12.2 for the BC and IC, respectively.

Second, to maintain a regular data schedule, the latencies of the QR cells are adjusted so that the x values and rotation parameters are output from the QR cells at the same time. The latency of the IC in producing these outputs can be expressed generically using a term L_{IC}. The latencies of the BC in producing the rotation parameters a and b are also set to L_{IC} to keep outputs synchronized. However, the latency of the BC in producing the δ_{out} is set to double this value, i.e. $2L_{\mathrm{IC}}$, as this relates back to the original scheduling of the full QR array, which showed that no two successive BC operations are performed on successive cycles. By keeping the structure of the data schedule, the retiming process comes down to a simple relationship, as illustrated in Table 12.3.

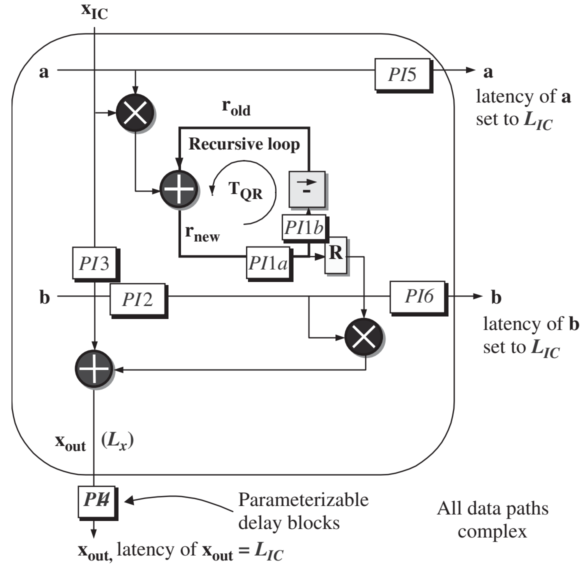


Figure 12.33 Generically retimed IC. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

Table 12.1 BC generic timing. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

BC	Delay value
B_{RL}	$2P_A + 2P_M + P_R + P_S - T_{QR}$
$PB1$	P_M
$PB2$	$T_{QR} - P_A - P_B$
$PB2a$	If $B_{RL} < 0$, then $-B_{RL}$, otherwise, 0
$PB2b$	If $B_{RL} < 0$, then $PB2 - PB2a$, otherwise $PB2$
$PB3$	If $B_{RL} > 0$, then B_{RL}, otherwise, 0
$PB4$	$2P_A + P_M + P_R + P_D - PB3$
$PB5$	$2P_A + 2P_M + P_R + P_D - T_{QR}$
$PB5a$	If $PB5 < 0$, then $PB5$, otherwise, 0
$PB5b$	If $PB5 > 0$, then $PB5$, otherwise, 0
$PB6$	$L_{IC} - L_{\delta}$
$PB7$	$L_{IC} - L_a$
$PB8$	$L_{IC} - L_b$

12.8.1 Retiming QR Architectures

This next section continues to discuss the retiming issues and how to include them in a generic architecture.

Table 12.2 IC generic timing. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

IC	Delay value
I_{RL}	$2P_A + P_M + P_R - T_{QR}$
$PI1$	$T_{QR} - P_A - P_S$
$PI1a$	If $I_{RL} < 0$, $-I_{RL}$, otherwise, 0
$PI1b$	If $I_{RL} < 0$, $PI1 - PI1a$, otherwise, $PI1$
$PI2$	If $I_{RL} > 0$, I_{RL}, otherwise, $PI1$
$PI3$	$PI2 + P_A + P_M$
$PI4$	$L_{IC} - L_x$
$PI5$	L_{IC}
$PI6$	$L_{IC} - PI2$

Table 12.3 Generic expressions for the latencies of the BC and IC. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

Latency	Value
L_a	P_M
L_b	$P_M + PB5$
L_δ	$PB3 + PB4 + 2P_M$
L_x	$PI3 + P_A$

Retiming of the Linear Array Architecture

The latency has the effect of stretching out the schedule of operations for each QR update. This means that iteration $n = 2$ begins $2m + 1$ clock cycles after the start of iteration $n = 1$. However, the introduction of processor latency stretches out the scheduling diagram such that the $n = 2$ iteration begins after $(2m + 1)L_{IC}$ clock cycles. This is obviously not an optimum use of the linear architecture as it would only be used every L_{IC}th clock cycle.

A factor, denoted as T_{QR}, was introduced in the last section as the number of cycles between the start of successive QR updates, as determined by the level of hardware reduction. It can be shown that a valid schedule which results in a 100% utilization can be achieved by setting the latency L_{IC} to a value that is relatively prime to T_{QR}. That is, if the two values do not share a common factor other than 1 then their lowest common multiple will be their product. Otherwise there will be data collisions at the products of L_{IC} and T_{QR} with their common multiplies.

$$\text{If}: T_{QR} = \bmod c \text{ and } L_{IC} = \bmod c$$

$$\text{Then}: T_{QR} = d \times c \text{ and } L_{IC} = e \times c$$

$$\text{Giving}: c = \frac{T_{QR}}{d} = \frac{L_{IC}}{e}$$

Where c is a common multiple of T_{QR} and L_{IC} and a positive integer other than 1, and, d and e are factors of T_{QR} and L_{IC}, respectively. Hence, there would be a collision at, $T_{QR} \times e = L_{IC} \times d$.

This means that the products of both $T_{QR} \times e$ and $L_{IC} \times d$ must be less than $T_{QR} \times L_{IC}$. Therefore, there is a collision of data. Conversely, to obtain a collision-free set of values c is set to 1.

The time instance $T_{QR} \times L_{IC}$ does not represent a data collision as the value of T_{QR} is equal to $2m + 1$, and the QR operation that was on line to collide with a new QR operation will have just been completed. The other important factor in choosing an optimum value of T_{QR} and L_{IC} is to ensure that the processors are 100% efficient.

The simple relationship between T_{QR} and L_{IC} is a key factor in achieving a high utilization for each of the types of structure. More importantly, the relationship gives a concise mathematical expression that is needed in the automatic generation of a generic QR architecture, complete with scheduling and retiming issues solved.

Figure 12.34 shows an example schedule for the 7-input linear array shown in Figure 12.17 where L_{IC} is 3 and T_{QR} is 7. The shaded cells represent the QR operations from different updates that are interleaved with each other and fill the gaps left by the highlighted QR update. The schedule is assured to be filled by the completion of the first QR update; hence, this is dependent on the latency L_{IC}.

12.9 Parameterizable QR Architecture

The main areas of parameterization include the wordlength, latency of arithmetic functions, and the value of T_{QR}. Different specifications may require different finite precision therefore the wordlength is an important parameter. The QR cells have been built up using a hierarchical library of arithmetic functions, which are parameterized in terms of wordlength, with an option to include pipelining to increase the operation speed as required. These parameters are passed down through the hierarchy of the HDL description of the QR cells to these arithmetic functions. Another consideration is the value of T_{QR}, which determines the length of the memory needed within the recursive loops of the QR cells which hold the R and u values from one QR update to the next. Both T_{QR} and the level of pipelining within the arithmetic functions are incorporated in generic timing expressions of the SGR QR cells.

12.9.1 Choice of Architecture

Table 12.4 demonstrates the process for designing a QR architecture when given a specific sample rate (100 MHz) and QR array size. The examples below are for a large QR array with 45 auxiliary inputs and 12 primary inputs, i.e. $m = 22$ and $p = 12$. The resulting processor array is $2m + 1 = 45$ rows by $m + p + 1 = 35$ columns. For a given sample throughput rate and clock rate we can determine the value for T_{QR}, as depicted in Table 12.4. Note that the resulting value for T_{QR}, and L_{IC} must be relatively prime, but for these examples we can leave this relationship at present.

The general description for T_{QR}, as shown above, can be rearranged to give the following relationship:

$$\frac{N_{IC}}{N_{rows}} = \frac{T_{QR}}{2m+1}$$

This result is rounded down to the nearest integer. There are 3 possibilities:

$$\text{If} \frac{T_{QR}}{2m+1} > 1 \text{then a sparse linear array is needed}$$

$$\text{If} \frac{T_{QR}}{2m+1} = 1 \text{then a linear array is needed}$$

$$\text{If} \frac{T_{QR}}{2m+1} < 1 \text{then a rectangular array is needed}$$

Figure 12.34 Schedule for a linear array with an IC latency of 3. *Reproduced from Linear QR Architecture for a Single Chip Adaptive Beamformer*, by G. Lightbody, R. Woods, R. Walke & J. McCanny, Journal of VLSI Signal Processing Systems 24, 67–81 2000, Kluwer Academic Publishers, Springer

Table 12.4 Example QR architectures

Architecture	No. of processors			T_{QR}	Data rate (MSPS)
	BC	IC	Total		
Full QR array (processor for each QR cell)	45	1530	1575	4	25
Rectangular 1 (12 linear arrays, one for 4 rows)	12	408	420	4	25
Rectangular 2 (3 linear arrays, one for 15 rows)	3	102	105	15	6.67
Sparse rectangular (2 columns of ICs to each processor of 3 linear arrays)	3	51	54	30	3.33
Linear (1 BC,26 ICs)	1	34	35	45	2.22
Sparse linear 2 columns of ICs to each IC processor on a linear array	1	17	18	90	1.11

The above equation is reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE.

Depending on the dimensions of the resulting architecture, the designer may decide to opt for a sparse rectangular architecture. Note that the maximum throughput rate that the full triangular array can meet for a 100 MHz clock is limited to the 25 MSPS due to the 4 cycle latency within the QR cell recursive path (i.e. 100 MHz $\div$ 4 $=$ 25 MSPS). The first rectangular array solution is meeting the same throughput performance as the full QR array while using only 408 ICs and 12 BCs instead of the full array which requires 1530 ICs and 45 BCs.

12.9.2 Parameterizable Control

A key aspect of the design of the various architectures is the determination of the control data needed to drive the multiplexers in these structures. Due to the various mappings that have been applied, it is more relevant of think of the IC operation as having four different modes of operation, input, mirrored input, unmirrored cell and mirrored cell (Figure 12.35, Walke 1997), where x represents the x-input data and θ represents the rotation parameters. The mirrored ICs are the result of the fold used to derive the rectangular processor array from the QR array, and simply reflect a different data flow. The cell orientation is governed by the multiplexers and control, and is therefore an issue concerning control signal generation.

The four modes of operation can be controlled using two control signals, C, which determines whether the x input is from the array (I) or from external data (E), and F, which distinguishes between a folded (M) and an unfolded operation (U). The latter determines the source direction of

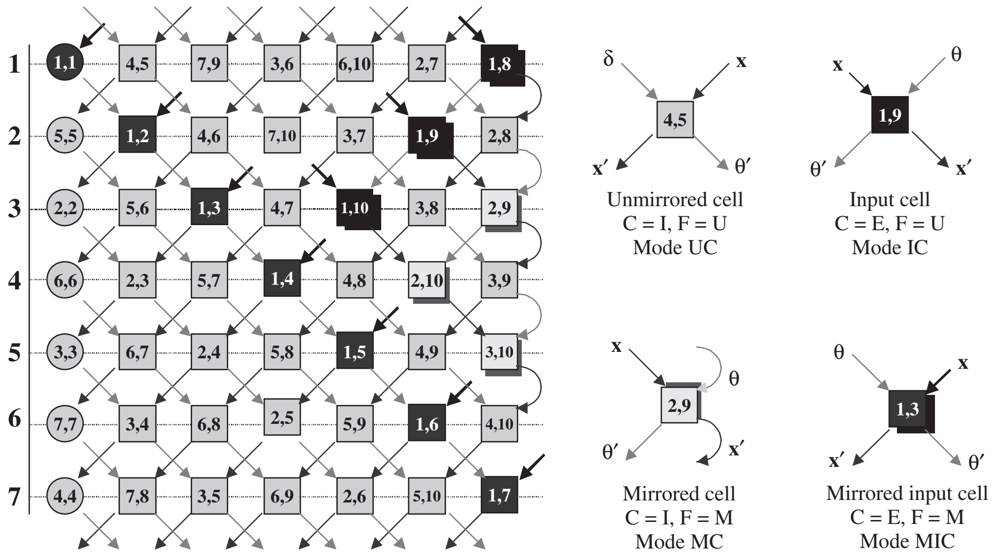


Figure 12.35 Types of cells in processor array. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

the inputs. The outputs are then from the opposite side of the cell. A mechanism for determining the control of each architecture is given in the next few sections.

12.9.3 Linear Architecture

The control signals for the linear architecture were derived directly from its data schedule. The modes of operation of the cells were determined for each cycle of the schedule, as shown in Table 12.5. Figure 12.36 shows the QR cells with the surrounding control and multiplexers needed to accept and process the correct data. The control signals for a full QR operation for this example are given in Table 12.6.

The control and timing of the architectures for the other variants is more complex, but can be derived from the original control for the linear array. Consideration needs to be given to the effect that latency has on the control sequences. In the sparse variants, extra delays need to be placed within the cells to organize the schedule, and in the rectangular variants, the cells need to be able to take x and θ inputs from the cells above and below, as well as from adjacent cells. Each of these variants shall be looked at in turn.

12.9.4 Sparse Linear Architecture

Figure 12.37(a) shows two columns of operations being assigned onto each IC. From the partial schedule shown in Figure 12.37(b) it can be seen that the transition of a value from left to right within the array requires a number of delays. The transfer of θ_1 from the BC(1,1), to the adjacent IC(1,2) takes three cycles. However, the transfer of X_{12} from the IC to the BC only takes one cycle.

Table 12.5 Modes of operation of the QR cells for the linear array

Cycle	BC1	IC2	IC3	IC4	IC5	IC6	IC7
1	IC	UC	UC	UC	UC	UC	MIC
2	UC	IC	UC	UC	UC	MIC	UM
3	UC	UC	IC	UC	MIC	UC	MIC
4	UC	UC	UC	IC	UM	MIC	UM
5	UC	UC	UC	UC	IC	UM	MIC
6	UC	UC	UC	UC	UM	IC	UM
7	UC	UC	UC	UC	UM	UM	IC

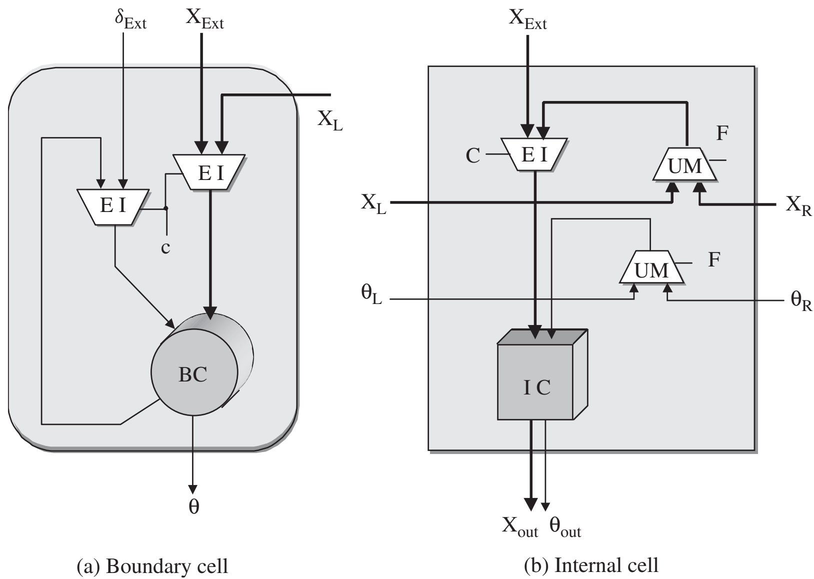


Figure 12.36 QR cells for the linear architecture. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

The example in Figure 12.38 shows the partitioning of three columns of ICs. Scheduling them onto a single processor requires their sequential order to be maintained. The IC operations have been numbered, 1, 2 and 3, for the first row, and $1'$, $2'$ and $3'$ for the second row. The outputs generated from operation 2 are required for operation $1'$ and $3'$. Because, all the operations are being performed on the same processor, then delays are needed to hold these values until they are required by operations $1'$ and $3'$. Operation 3 is performed before operation $1'$, and operations 3, $1'$, and $2'$ are performed before operation $3'$, which relates to two and four cycle delays respectively.

Table 12.6 Linear array control for the external/internal x inputs and for mirrored / not mirrored cells

Cycle	C1	C2	C3	C4	C5	C6	C7	F1	F2	F3	F4	F5	F6
1	E	I	I	I	I	I	E	U	U	U	U	U	M
2	I	E	I	I	I	E	I	U	U	U	U	M	U
3	I	I	E	I	E	I	I	U	U	U	M	U	M
4	I	I	I	E	I	I	I	U	U	U	U	M	U
5	I	I	I	I	E	I	I	U	U	U	U	U	M
6	I	I	I	I	I	E	I	U	U	U	U	U	U
7	I	I	I	I	I	I	E	U	U	U	U	U	U
8	E	I	I	I	I	I	E	U	U	U	U	U	M
9	I	E	I	I	I	E	I	U	U	U	U	M	U
10	I	I	E	I	E	I	I	U	U	U	M	U	M
11	I	I	I	E	I	I	I	U	U	U	U	M	U
12	I	I	I	I	E	I	I	U	U	U	U	U	M
13	I	I	I	I	I	E	I	U	U	U	U	U	U

This has been generically defined according to the number of columns of operations within the processor array assigned to each IC, N_{IC}, as shown in Figure 12.39.

In general terms, the two output values x and θ are transferred from operation $(c+1)$ to c and $(c+2)$ to match Figure 12.39. The value that is fed to a specific operation depends on whether the cells perform the folded or unfolded modes of operation as summarized in Table 12.6. If the data is transferred between the same type of cell (i.e. $U \rightarrow U$, or $M \rightarrow M$) then the delay will be either $N_{IC}-1$ or $N_{IC}+1$, according to Table 12.7. However, if the data transfer is between different types of cell (i.e. $U \rightarrow M$, or $M \rightarrow U$, as in the case of the end processor), then the number of delays will be N_{IC}. This is summarized in Table 12.7.

Table 12.7 Required delays for sparse linear array

Transfer of data between cells: U not mirrored; M mirrored	Direction in terms of QR operation	Direction of data flow shown in processor array	Delays needed	Label
$U \rightarrow U$	$(i, j) \rightarrow (i, j+1), \theta$	$\rightarrow$	$N_{IC}+1$	D1
	$(i, j) \rightarrow (i+1, j), x$	$\leftarrow$	$N_{IC}-1$	D2
$M \rightarrow M$	$(i, j) \rightarrow (i+1, j), \theta$	$\leftarrow$	$N_{IC}-1$	D2
	$(i, j) \rightarrow (i, j+1), x$	$\rightarrow$	$N_{IC}+1$	D1
$U \rightarrow M$(end cell)	$(i, j) \rightarrow (i, j+1), \theta$	$\downarrow$	N_{IC}	D3
$M \rightarrow U$(end cell)	$(i, j) \rightarrow (i+1, j), x$	$\downarrow$	N_{IC}	D3

Figure 12.37 Sparse linear array schedule

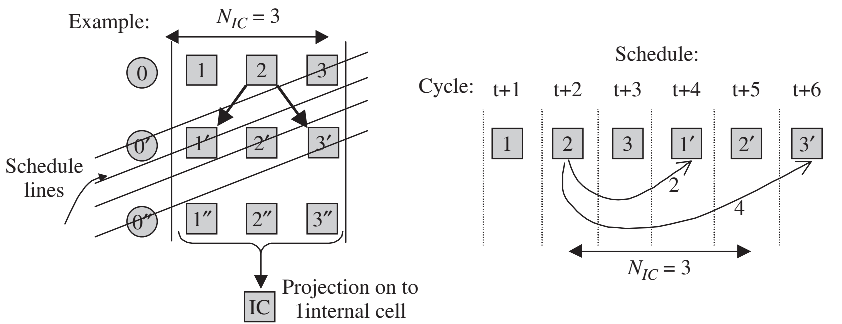


Figure 12.38 Example partitioning of three columns onto one processor

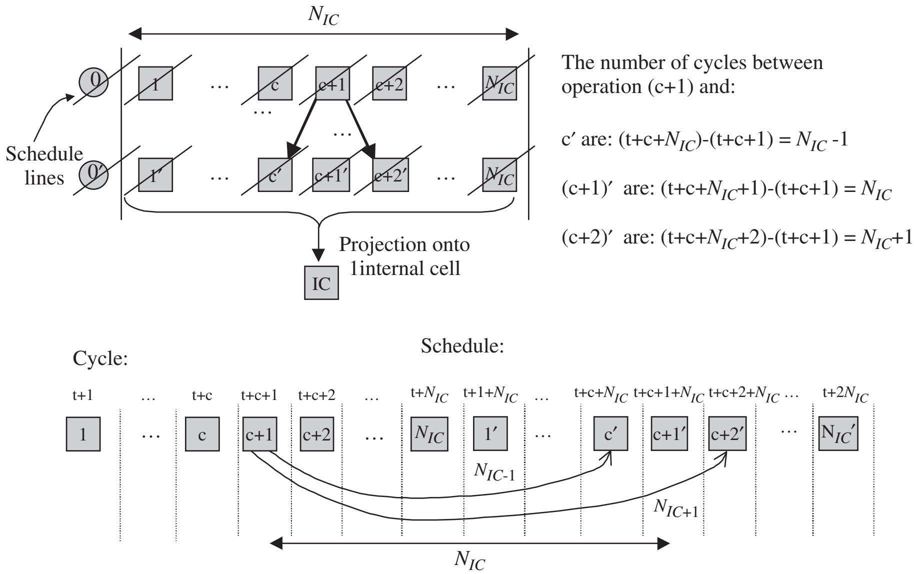


Figure 12.39 Generic partitioning of N_{IC} columns onto one processor

These delays are then used within the sparse linear architecture to keep the desired schedule, shown in Figure 12.40. The three levels of delays are denoted by the square blocks labelled D1, D2 and D3. These delays can be redistributed to form a more efficient QR cell architecture, as shown in Figure 12.41. The extra L and R control signals indicate the direction source of the inputs, with E and I control values determining whether the inputs come from an adjacent cell or from the same cell. EC refers to the end IC that differs slightly in that there are two modes of operation when the cell needs to accept inputs from its output. The control sequences for this example are given in Table 12.8.

From Table 12.8, it can be seen that EC is the same as R and is the inverse of L. In addition, the states alternate between E and I with every cycle, therefore, one control sequence could be used to determine the control of the internal inputs. This control value has been labeled D. The control signals are categorized as external input control, C_i, fold control F_i, array control L_i and internal input control D_i The subscripts are coordinates representing the cells to which the control signals are being fed.

One of the key issues with the sparse linear array is the effect of the latencies in the QR cells on the schedule (which previously assumed a one cycle delay). With the linear architecture, the schedule was scaled by the latency. However, with the sparse linear array there was a concern that the delays $N_{\mathrm{IC}} - 1$, N_{IC}, and $N_{\mathrm{IC}} + 1$, would also need to be scaled in order to keep the structure of the original schedule, which would cause inefficiency.

In the example given in Figure 12.42, the latency of the IC is 3, so this gives the minimum value for N_{IC} as 4. $N_{\mathrm{IC}} + 1$ is therefore 5 and $N_{\mathrm{IC}} - 1$ is 3 clock cycles. The shaded cells in Figure 12.42 show one complete QR update with interconnection included. The rest of the QR operations are shown, but with limited detail to aid clarity. Since it is most probable that the latency of the IC will exceed the number of columns assigned to each processor, then it figures that the delays within

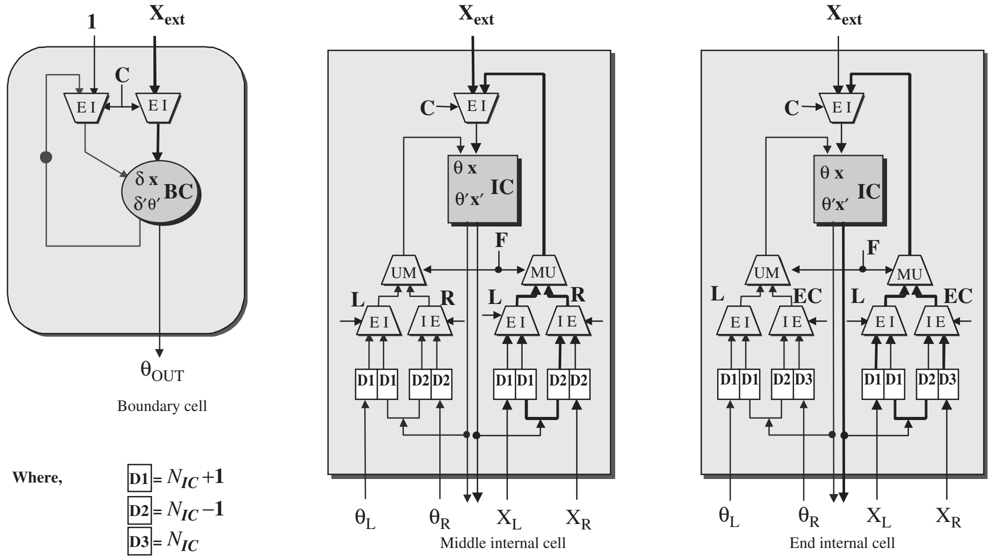


Figure 12.40 Sparse linear array cells

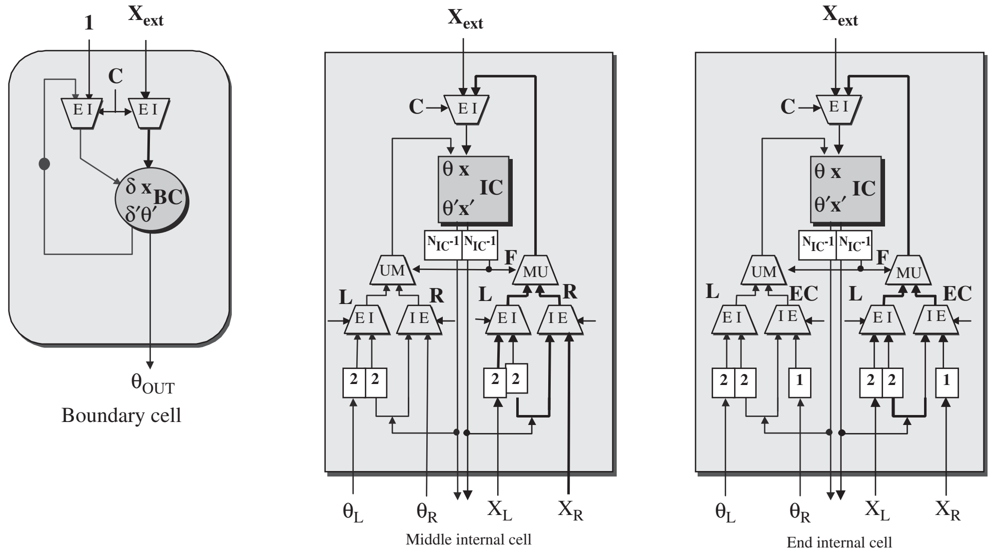


Figure 12.41 Redistributed delays for sparse linear array cells

Table 12.8 Control sequence for sparse linear array.

Cycle	External input	External input control				Fold control			Internal input control		
		C1	C2	C3	C4	F2	F3	F4	L	R	EC
1	$X_1(1)$	E	I	I	I	U	U	M	E	I	E
2	$X_8(0)$	I	I	I	E	U	U	U	I	E	I
3		I	I	I	I	U	U	U	E	I	E
4	$X_2(1)$	I	E	I	I	U	U	U	I	E	I
5	$X_7(0)$	I	I	I	E	U	U	U	E	I	E
6		I	I	I	I	U	U	U	I	E	I
7	$X_3(1)X_8(0)$	I	E	I	E	U	U	M	E	I	E
8	$X_9(0)$	I	I	I	E	U	U	M	I	E	I
9	$X_{10}(0)$	I	I	E	I	U	M	U	E	I	E
10	$X_4(1)$	I	I	E	I	U	U	U	I	E	I
11		I	I	I	I	U	U	M	E	I	E
12		I	I	I	I	U	U	M	I	E	I
13	$X_5(1)$	I	I	E	I	U	U	U	E	I	E
14		I	I	I	I	U	U	U	I	E	I

the linear sparse array will depend on L_{IC}, i.e. the $(N_{IC} - 1)$ delay will not be needed and the schedule realignment will be performed by the single and double cycle delays shown by numbered boxes in Figure 12.41. The highlighted cells represent a full QR update while the other numbered cells represent interleaved QR operations. The faded gray BCs with no numbers represent unused positions within the schedule.

12.9.5 *Rectangular Architecture*

The rectangular architecture consists of multiple linear array architectures that are concatenated. Therefore, the QR cells need to be configured so that they can accept inputs from the above linear array. In addition, the top linear array needs to be able to accept values from the bottom linear array. The QR cells are depicted in Figure 12.43. The control signals, E and I, decide on whether the X inputs are external (i.e. system inputs) or internal. The control value T refers to inputs from the above array and A refers to inputs from adjacent cells. When used as subscripts, TR and TL refer to values coming from the left and right cells of the array above; AR and AL refer to the values coming from the right and left adjacent cells within the same linear array.

12.9.6 *Sparse Rectangular Architecture*

The QR cells for the sparse rectangular array need to be able to feed inputs back to themselves in addition to the variations already discussed with the linear and rectangular architectures. The extra control circuitry is included in the QR diagrams shown in Figure 12.44. The control and the delays required by the sparse arrays to realign the schedule are brought together into LMR multiplexer cells (Figure 12.45) that include delays needed to take account of the retiming analysis demonstrated in this section.

It was discussed in connection with the sparse linear array how certain transfer in data values required the insertion of specific delays to align the schedule. This also applies to the rectangular array and the same rules can be used.

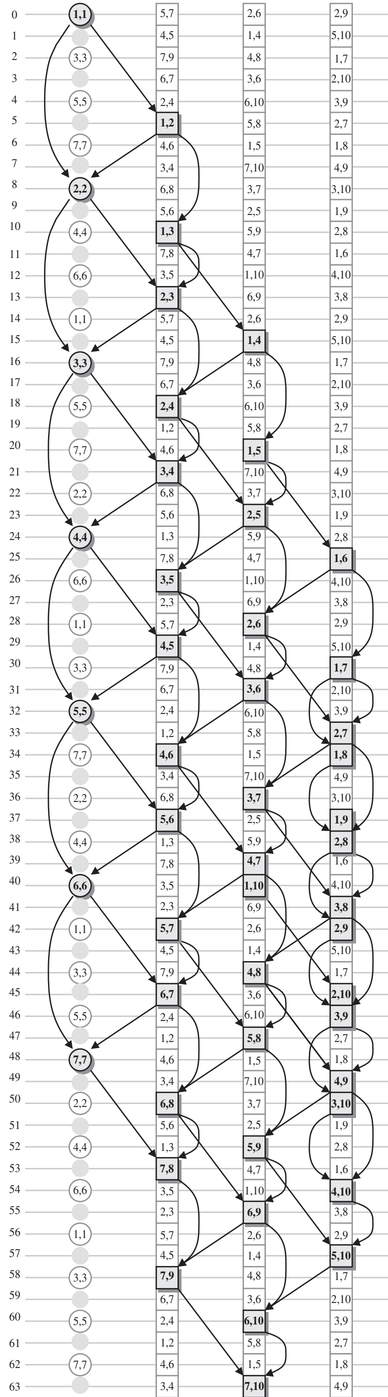


Figure 12.42 Effect of latency on schedule for the sparse linear array ($L_{\mathrm{IC}} = 3$)

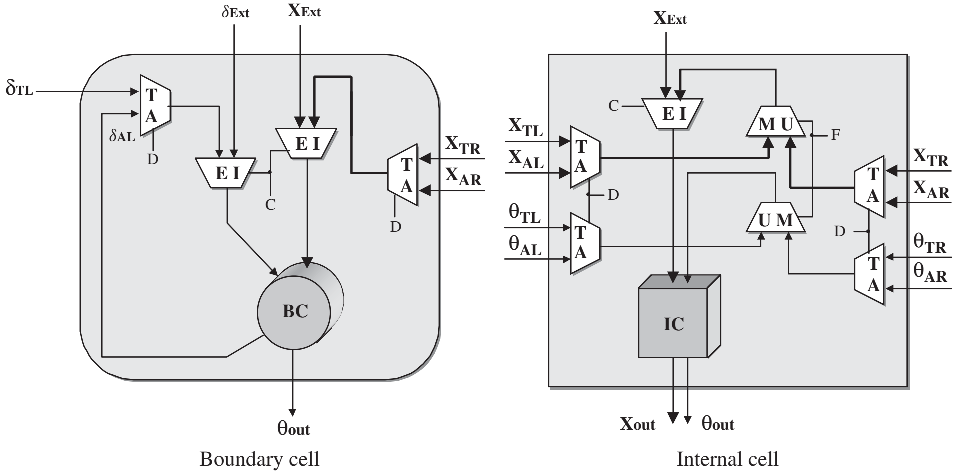


Figure 12.43 QR cells for the rectangular array

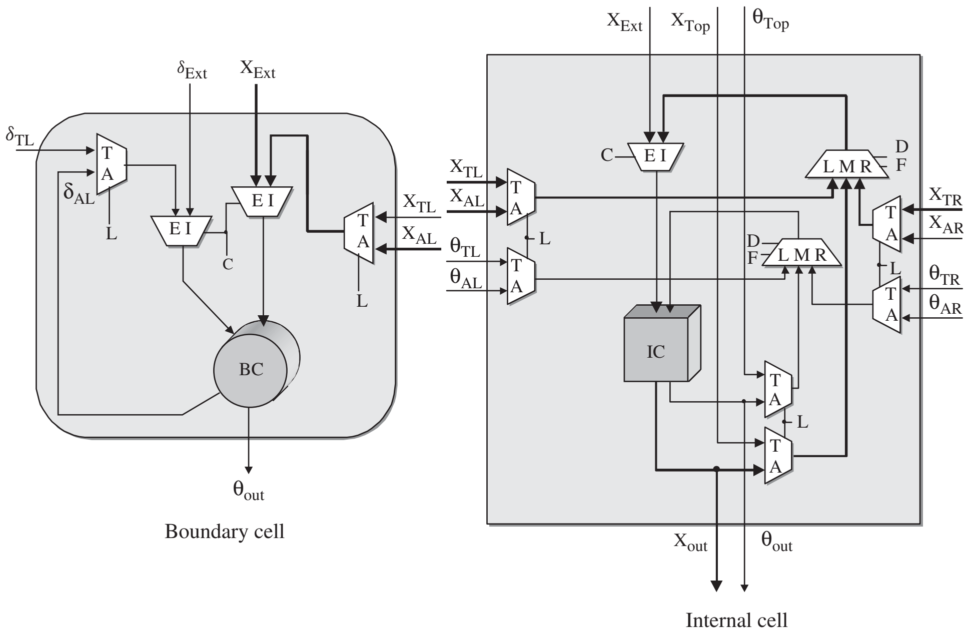


Figure 12.44 QR cells for the sparse rectangular array. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

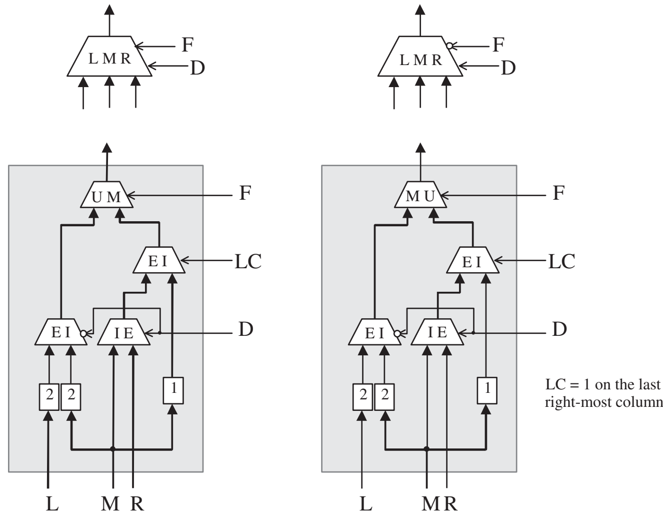


Figure 12.45 LMR control circuitry for sparse arrays. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

The starting point for determining the schedule for the sparse rectangular array is the schedule for the sparse linear array. From this, the rows of operations are divided into sections, each to be performed on a specific sparse linear array. The control, therefore, is derived from the control for the sparse linear version.The next section deals with parametric ways of generating the control for the various QR architectures. In addition to the control shown so far, the next section analyses how latency may be accounted for within the control generation.

12.9.7 Generic QR Cells

The sparse rectangular array QR cells shown in Figure 12.44 can be used for all the QR architecture variants, by altering the control signals and timing parameters. However, in the sparse variants there are added delays embedded within the LMR control cells. These can be removed for the full linear and rectangular array versions, by allowing them to be programmable so that they may be set to zero for the non-sparse versions. The key to the flexibility in the parameterizable QR core design is the generic generation of control signals. This is discussed in the following section.

12.10 Generic Control

The previous section detailed the various architectures derived from the QR array. Some detail was given of the control signals needed to operate the circuits. This section looks at generic techniques of generating the control signals that may be applied to all the QR architecture variants. It is suggested that a software interface is used to calculate each control sequence as a bit-vector seed (of length T_{QR}) that may be fed through a linear feedback register which will allow this value to be cyclically output bit by bit to the QR cells. The first stage in developing the control for a sparse QR array architecture is to look at the generic processor array which gives the control needed for the linear array. This can be used as a base from which to determine the control by folding and

Cycle	Input	Lineary array							Input	Sparse linear array			
		C1	C2	C3	C4	C5	C6	C7		C1	C2	C3	C4
1	$X_1(1)$, $X_8(0)$	E	I	I	I	I	I	E	$X_1(1)$	E	I	I	I
2	$X_2(1)$, $X_9(0)$	I	E	I	I	I	E	I	$X_6(0)$	I	I	I	E
3	$X_3(1)$, $X_{10}(0)$	I	I	E	I	E	I	I		I	I	I	I
4	$X_4(1)$,	I	I	I	E	I	I	I	$X_2(1)$	I	E	I	I
5	$X_5(1)$,	I	I	I	I	E	I	I	$X_7(0)$	I	I	I	E
6	$X_6(1)$,	I	I	I	I	I	E	I		I	I	I	I
7	$X_7(1)$,	I	I	I	I	I	I	E	$X_3(1)$, $X_8(0)$	I	E	I	E
8	$X_8(1)$, $X_1(2)$	E	I	I	I	I	I	E	$X_9(0)$	I	I	I	E
9	$X_9(1)$, $X_2(2)$	I	E	I	I	I	E	I	$X_{10}(0)$	I	I	E	I
10	$X_{10}(1)$,$X_3(2)$	I	I	E	I	E	I	I	$X_4(1)$	I	I	E	I
11	$X_4(2)$	I	I	I	E	I	I	I		I	I	I	I
12	$X_5(2)$	I	I	I	I	E	I	I		I	I	I	I
13	$X_6(2)$	I	I	I	I	I	E	I	$X_5(1)$	I	I	E	I
14	$X_7(2)$	I	I	I	I	I	I	E		I	I	I	I

Figure 12.46 Control for the external inputs for the linear QR arrays. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

manipulating the control signals sequence into the required sequence for the sparse linear arrays. The control for the rectangular versions may be generated quite simply from the control for the linear architectures.

12.10.1 Generic Input Control for Linear and Sparse Linear Arrays

The input of external X values follows the original folding of the QR triangular array with a new external X-input fed into a cell of the linear array on each cycle. It starts from the left-most cell, reaching the right-most cell and then folding back until all the $2m + p + 1$ inputs are fed into the array for that specific QR update. This is highlighted for one set of QR inputs in Figure 12.46. The next set of inputs follow the same pattern, but start after T_{QR} cycles. This results in a repetitive segment of control that repeats every T_{QR} cycles, (which is 7 for the linear array example and 14 for the sparse linear array example). From this, it can be possible to automatically generate the control vectors, containing T_{QR} bits, which represent the repeating sections for each of the control signals, C1–C7. Determining the start of a set of QR inputs in relation to the previous set can be achieved using the dimensions of the original array, and the resulting processor array from which the reduced QR architectures are derived. This relationship is depicted by Figure 12.47. The heavy lines indicate the series of inputs for one QR update, and relate to the highlighted control for the external inputs for the linear array example in Figure 12.46.

Software code can be written to generate the control signals for the external inputs for the linear and sparse linear array. The inputs are broken down into two series (Figure 12.47), one dealing with the inputs going from left to right, and the other dealing with the inputs from right to left (the change in direction being caused by the fold). The code generates the position of the control signals within the control vector for each input into each processor. If the vector number is larger than the vector then the vector size is subtracted from this value, leaving the modulus as the position. However, after initializing the operation of the QR array it is necessary to delay this control signal by an appropriate value.

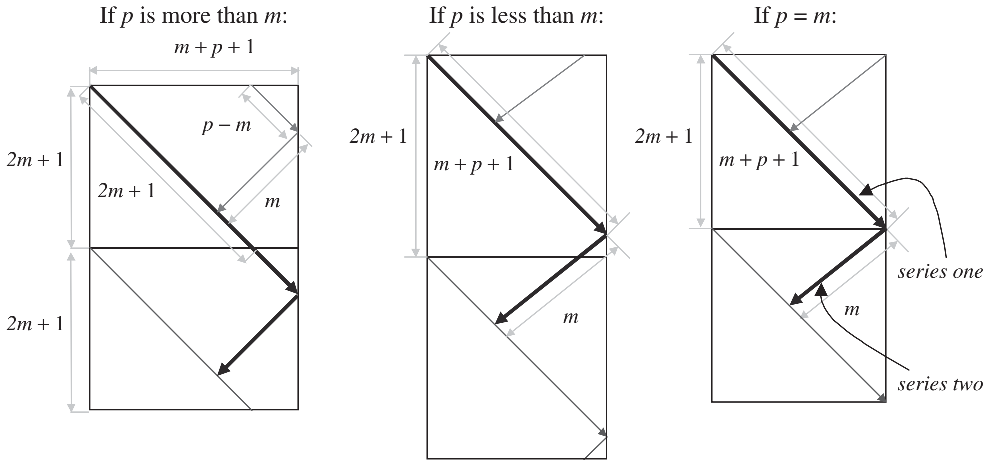


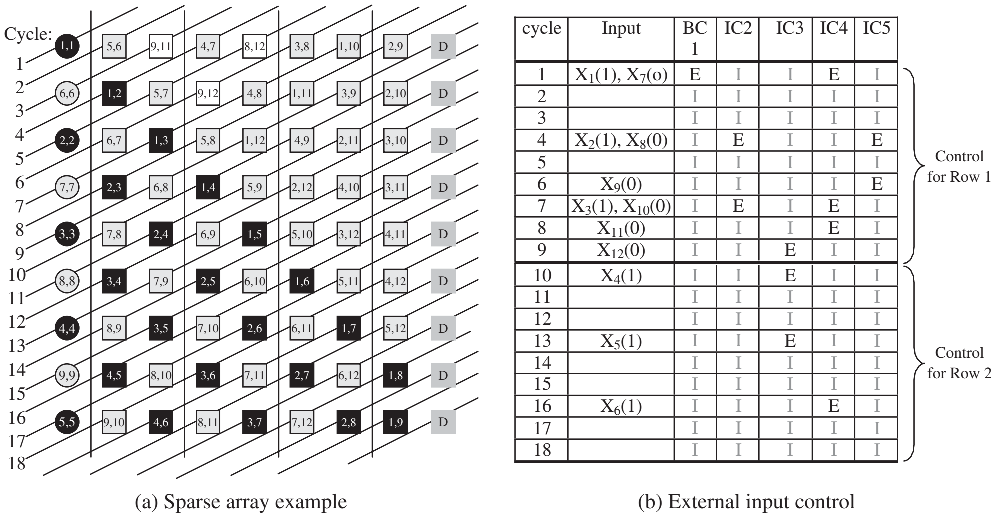
Figure 12.47 Control for the external inputs for the linear QR arrays. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

12.10.2 Generic Input Control for Rectangular and Sparse Rectangular Arrays

The control from the rectangular versions is relatively easy and comes directly from the control vectors for the linear array, given in Figure 12.46, by dividing the signals vectors into parts relating to the partitions within the processor array. For example, if the control seed vectors for the linear array are 8 bits wide and the rectangular array for the same system consists of two rows, then each control vector seeds would be divided into two 4 bit wide vectors. One for the first rectangular array and the other for the second. The control seed for the sparse rectangular array is derived in the same manner, but from the control of the sparse linear array with the same value of N_{IC}. The same code may be edited to include the dummy operations that may be required for the sparse versions. Figure 12.48(a) shows an example sparse linear array mapping with $m = 4$, $p = 3$, and $N_{\mathrm{IC}} = 2$. The control (Figure 12.48b) can be divided into two sections for implementing a sparse rectangular array consisting of two rows of the sparse linear array.

12.10.3 Effect of Latency on the Control Seeds

The next stage is to determine the effect that latency has on the control vectors. Previously delay values, $D1$, $D2$ and $D3$, as discussed for the sparse linear array, were necessary to account for multiple columns (N_{IC}) of operations applied to each IC, where, for a system with a single cycle latency, $D1 = (N_{IC} - 1)$, $D2 = N_{IC}$, and $D3 = (N_{IC} + 1)$. However, in the real system the processors have multiple latency. It is assumed that the latency of the IC (L_{IC}) will be greater than these delays the latencies are increased such that the appropriate delays become $D1 = L_{\mathrm{IC}}$, $D2 = L_{\mathrm{IC}} + 1$, and $D3 = L_{\mathrm{IC}} + 2$. For the linear array the values $D1$, $D2$ and $D3$ are all set to L_{IC}; then the code may be used to generate the control vectors. The only difference is when the position of the control value exceeds the width of the vector. With the single latency versions this was accounted for by subtracting the value T_{QR} from the value, (where the width of the vector seed is T_{QR}).



cycle	Input	BC 1	IC2	IC3	IC4	IC5
1	$X_1(1)$, X_7(o)	E	I	I	E	I
2		I	I	I	I	I
3		I	I	I	I	I
4	$X_2(1)$, $X_8(0)$	I	E	I	I	E
5		I	I	I	I	I
6	$X_9(0)$	I	I	I	I	E
7	$X_3(1)$, $X_{10}(0)$	I	E	I	E	I
8	$X_{11}(0)$	I	I	I	E	I
9	$X_{12}(0)$	I	I	E	I	I
10	$X_4(1)$	I	I	E	I	I
11		I	I	I	I	I
12		I	I	I	I	I
13	$X_5(1)$	I	I	E	I	I
14		I	I	I	I	I
15		I	I	I	I	I
16	$X_6(1)$	I	I	I	E	I
17		I	I	I	I	I
18		I	I	I	I	I

(a) Sparse array example (b) External input control

Figure 12.48 Partitioning of control seed. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

Cycle	Input		c1	c2	c3	c4	c5	c6	c7	c8	c9
1	$X_1(1)$	$X_8(0)$	E	I	I	I	I	I	I	E	I
2	$X_2(1)$	$X_9(0)$	I	E	I	I	I	I	I	I	E
3	$X_3(1)$	$X_{10}(0)$	I	I	E	I	I	I	I	I	E
4	$X_4(1)$	$X_{11}(0)$	I	I	I	E	I	I	I	E	I
5	$X_5(1)$	$X_{12}(0)$	I	I	I	I	E	I	E	I	I
6	$X_6(1)$		I	I	I	I	I	E	I	I	I
7	$X_7(1)$		I	I	I	I	I	I	E	I	I

Figure 12.49 Control when latency = 1. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

Finding the section that repeats is complicated as the delays lengthen the control sequence. However, a repetitive control sequence of length T_{QR} can still be found. When latency is included within the calculations, it is not sufficient to reduce the value to within the bounds of the vector width. Alternatively, the position of the control value within the vector is found by taking the modulus of T_{QR}. An analysis of the effect of latency on the control vectors is shown through an example linear array where $m = 3$, $p = 5$ and $T_{QR} = 2m + 1 = 7$. Figure 12.49 shows the control vectors for the system with single cycle latency. Figure 12.50, shows the effect of the control timing for the set of inputs for one QR update when a latency of 3 is considered. Using these control value positions gives the control vectors shown in Figure 12.51.

One point to highlight is the fact that there may be several cycles of the control vector before the required input is present. For example, the vector in the above example for C4 is [I I E I I I I], however, the first required input is at time=10, not 3. Therefore it is necessary to delay the start of this control signal by 7 cycles. The technique relies on the use of initialization control

Input	Latency = 1		Latency = 3	
	Time	Position in vector	Time	Position in vector
X_1	1	1	1	1
X_2	2	2	4	4
X_3	3	3	7	7
X_4	4	4	10	10mod7 = 3
X_5	5	5	13	13mod7 = 6
X_6	6	6	16	16mod7 = 2
X_7	7	7	19	19mod7 = 5
X_8	8	8mod7 = 1	22	22mod7 = 1
X_9	9	9mod7 = 2	25	25mod7 = 4
X_{10}	10	10mod7 = 3	28	28mod7 = 0, set to 7
X_{11}	11	11mod7 = 4	31	31mod7 = 3
X_{12}	12	12mod7 = 5	34	34mod7 = 6

Figure 12.50 Determining the position in control vector. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

cycle	Inpu t				c1	c2	c3	c4	c5	c6	c7	c8	c9
1		$X_8(1)$		$X_1(4)$	E	I	I	I	I	I	I	E	I
2		$X_6(2)$			I	I	I	I	I	E	I	I	I
3	$X_{11}(0)$		$X_4(3)$		I	I	I	E	I	I	I	E	I
4		$X_9(1)$		$X_2(4)$	I	E	I	I	I	I	I	I	E
5		$X_7(2)$			I	I	I	I	I	I	E	I	I
6	$X_{12}(0)$		$X_5(3)$		I	I	I	I	E	I	E	I	I
7		$X_{10}(1)$		$X_3(4)$,	I	I	E	I	I	I	I	I	E

Figure 12.51 Effect of latency on control seeds. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

signals to start the cycling of the more complicated control vectors for the processors. This would be provided by the overall system level control for the full application. The method discussed offers a parametric way of dealing with control signal generation and allows the majority of the control to be localized. These same principles applied to the control signals for the timing of the external inputs may be extended for the rest of the control signals, i.e. the fold, internal input, and row control.

12.11 Beamformer Design Example

For a typical beamforming application in radar, the values of m would be in range of 20 to over 100. The number of primary inputs p would typically range from 1 to 5 for the same application. An example specification is given in Figure 12.52. One approach is to use the QR array. Assuming the fastest possible clock rate, f_{CLK}, then the fundamental loop will dictate the performance for example, resulting in a design with 25% utilisation for a T_{QR} of four clock cycles. Thus the major challenge is now to select the best architecture which will closest match the throughput rate with the best use of hardware. For the example here, a desired input sample rate of 15 MSPS with a maximum possible clock rate of 100 MHz is assumed.

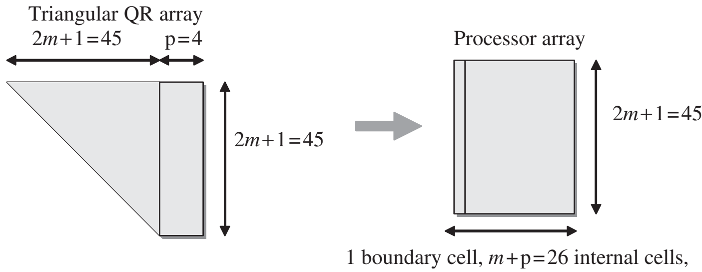


Figure 12.52 Example QR architecture derivation, $m = 22$, $p = 4$. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

The value for T_{QR} can be calculated using the desired sample rate S_{QR} and the maximum clock rate f_{CLK}:

$$T_{QR} = \frac{f_{CLK}}{S_{QR}} = \frac{100 \times 10^6}{15 \times 10^6} = 6.67$$

This value is the maximum number of cycles allowed between the start of successive QR updates, therefore, it needs to be rounded down to the nearest integer. The ratio of N_{rows}/N_{IC} can be obtained by substituting for the known parameters into the relationship below:

$$\frac{N_{rows}}{N_{IC}} = \frac{2m+1}{T_{QR}} = \frac{45}{6} = 7.5$$

where $1 \leq N_{rows} \leq 2m + 1$ (i.e. 45) and $1 \leq N_{IC} \leq m + p$ (i.e. 26). Using these guidelines an efficient architecture can be derived by setting $N_{IC} = 2$, and hence, $N_{rows} = 15$. The operations are distributed over 15 sparse linear architectures, each with 1 BC and 13 ICs, as shown in Figure 12.53.

Also note that the critical path within the circuit must be considered to ensure that the core can be clocked fast enough to support the desired QR operation. Here, additional pipeline stages may be added to reduce the critical path and therefore improve the clock rate. However, this has the

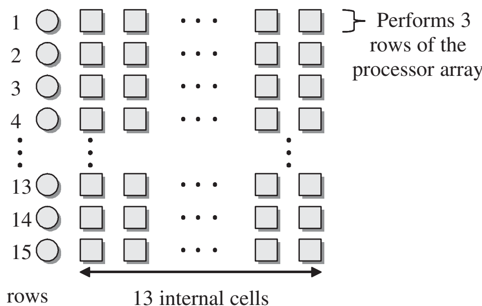


Figure 12.53 Example architecture. Reproduced from *Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering*, by G. Lightbody & R. Woods, IEEE Trans on VLSI Systems, Vol. 11, No. 4, © 2003 IEEE

effect of increasing the latencies and these must then be included in the architecture analysis. Each row of processors is responsible for three rows of operations within the processor array, therefore, $T_{QR} = 6$, resulting in an input sample rate of 16.67 MSPS, which exceeds the required performance. The details of some example architectures for the same QR array are given in Table 12.4.

The value for T_{QR} for the full QR array implementation is determined by the latency in the recursive loop of the QR cells (consisting of a floating-point addition and a shift-subtract function). For the example shown, the QR array needs to wait four clock cycles for the calculation of the value in the recursive loop, which therefore determines the sample rate of the system. This example emphasizes the poor return of performance of the full QR implementation at such a high cost of hardware. As shown in Table 12.4 the same performance may be achieved by using a rectangular array of a reduced number of processors resulting in a saving in hardware resources at no cost.

12.12 Summary

The goal of this chapter was to document each of the stages of the development for an IP core for adaptive beamforming. The main aspects covered were the design choices made with regard to:

- decision to use design for reuse strategies to develop an IP core
- determination of the algorithm
- determination of a suitable component to design as an IP core
- specifying the generic parameters
- algorithm to architecture development
- scalable architectures
- scalable scheduling of operations and control

Each stage listed above was detailed for the adaptive beamforming example. Background information was supplied regarding the RLS choice of algorithm decided upon for the adaptive weight calculations. The key issue with the algorithm used is its computational complexity. Techniques and background research were summarized, showing the derivation of the simplified QR-RLS algorithm suitable for implementation on a triangular systolic array. Even with such reduction in the complexity, there may still be a need to map the full QR-array down onto a reduced architecture set. This formed a key component of the chapter giving a step-by-step overview of how such a process can be achieved while maintaining a generic design. Focus was given to architecture scalability and the effects of this on operation scheduling. Further detail was given to the effects of processor latency and retiming on the overall scheduling problem, showing how such factors could be accounted for upfront. Finally, examples were given on how control circuitry could be developed so to scale with the architecture, while maintaining performance criteria. It is envisaged that the principles covered by this chapter should be expandable to other IP core developments.

References

Athanasiadis T, Lin K and Hussain Z (2005) Space-time OFDM with adaptive beamforming for wireless multimedia applications. *ICITA 2005, 3rd Int. Conf. Information Technology and Applications*, pp. 381–386.

Baxter P and McWhirter J (2003) Blind signal separation of convolutive mixtures. *Conference Record of the 37th Asilomar Conference on Signals, Systems and Computers*, pp. 124–128.

Bitmead R and Anderson B (1980) Performance of adaptive estimation algorithms in dependent random environments. *IEEE Transactions on Automatic Control*, **25**(4), 788–794.

Choi S and Shim D (2000) A novel adaptive beamforming algorithm for a smart antenna systemin a CDMA mobile communication environment. *IEEE Transactions on Vehicular Technology*, **49**(5), 1793–1806.

Cioffi J (1990) The fast Householder filters-RLS adaptive filter. *ICASSP-90. International Conference on Acoustics, Speech, and Signal Processing,* pp. 1619–1622.

Cioffi J and Kailath T (1984) Fast, recursive-least-squares transversal filters for adaptive filtering. *Acoustics, Speech, and Signal Processing* see also *IEEE Transactions on Signal Processing* **32**(2), 304–337.

de Lathauwer L, de Moor B and Vandewalle J (2000) Fetal electrocardiogram extraction by blind source subspace-separation. *IEEE Transactions on Biomedical Engineering* **47**(5), 567–572.

Döhler R (1991) Squared Givens rotation. *IMA Journal of Numerical Analysis* **11**(1), 1.

Eleftheriou E and Falconer D (1986) Tracking properties and steady-state performance of RLS adaptive filter algorithms. *Acoustics, Speech, and Signal Processing*; see also *IEEE Transactions on Signal Processing* **34**(5), 1097–1110.

Eweda E (1998) Maximum and minimum tracking performances of adaptive filtering algorithms over target weight cross correlations. *Circuits and Systems II: Analog and Digital Signal Processing*, see also *IEEE Transactions on Circuits and Systems* **45**(1), 123–132.

Eweda E and Macchi O (1987) Convergence of the RLS and LMS adaptive filters. *IEEE Transactions on Circuits and Systems*, **34**(7), 799–803.

Gentleman W and Kung H (1981) Matrix triangularization by systolic arrays. *Proc. SPIE* **298**, 19–26.

Givens W (1958) Computation of plane unitary rotations transforming a general matrix to triangular form. *Journal of the Society for Industrial and Applied Mathematics* **6**(1), 26–50.

Hamill R (1995) VLSI algorithms and architectures for DSP arithmetic computations, PhD Thesis, Queen's University Belfast.

Haykin S (2002) *Adaptive Filter Theory*. Beijing: Publishing House of Electronics Industry, pp. 320–331.

Hsieh S, Liu K and Yao K (1993) A unified approach for QRD-Based recursive least-squares estimation without square roots. *IEEE Transaction on Signal Processing* **41**(3), 1405–1409.

Hudson J (1981) *Adaptive Array Principles*. Institution of Engineering and Technology.

Kalouptsidis N and Theodoridis S (1993) *Adaptive System Identification and Signal Processing algorithms*. Prentice-Hall, Upper Saddle River, NJ, USA.

Lightbody G (1999) High performance VLSI architectures for recursive least squares adaptive filtering, PhD Thesis Queen's University Belfast.

Lightbody G, Woods R and Francey J (2007) Soft IP core implementation of recursive least squares filter using only multiplicative and additive operators *Proc. Int. Conf. on Field Programmable Logic*, pp. 597–600.

Lightbody G, Woods R and Walke R (2003) Design of a parameterizable silicon intellectual property core for QR-based RLS filtering. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **11**(4), 659–678.

Liu K, Hsieh S and Yao K (1990) Recursive LS filtering using block Householder transformations. *Proc. Int. Acoustics, Speech, Signal Processing Conf.*, pp. 1631–1634.

Liu K, Hsieh S and Yao K (1992) Systolic block Householder transformation for RLS algorithm with two-level pipelined implementation. *IEEE Transactions on Signal Processing* **40**(4), 946–958.

McCanny J, Ridge D, Hu Y and Hunter J (1997) Hierarchical VHDL Libraries for DSP ASIC Design. *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Proc.*, p. 675.

McWhirter J (1983) Recursive least-squares minimization using a systolic array. *Proc. SPIE* **431**, 105–109.

Morgan D and Kratzer S (1996) On a class of computationally efficient, rapidly converging, generalized NLMS algorithms. *Signal Processing Letters, IEEE* **3**(8), 245–247.

Rader C (1992) MUSE-a systolic array for adaptive nulling with 64 degrees of freedom, using Givens transformations and wafer scale integration. *Proc. Int. Conf. on Application Specific Array Processors*, pp. 277–291.

Rader C (1996) VLSI systolic arrays for adaptive nulling. *Signal Processing Magazine, IEEE* **13**(4), 29–49.

Rader C and Steinhardt A (1986) Hyperbolic householder transformations. *IEEE Transactions on Signal Processing* **34**(6), 1589–1602.

Shan T and Kailath T (1985) Adaptive beamforming for coherent signals and interference. *IEEE Transactions on Signal Processing* **33**(3), 527–536.

Shepherd T and McWhirter J (1993) Systolic adaptive beamforming. *Array Signal Processing* pp. 153–243.

Tamer O and Ozkurt A (2007) Folded systolic array based MVDR beamformer. *ISSPA 2007, International Symposium on Signal Processing and its Applications, Sharjah United Arab Emirates*.

University N (2007) Variable precision floating point modules. Web publication downloadable from http://www.ece.neu.edu/groups/rpl/projects/floatingpoint/index.html.

Walke R (1997) High sample rate givens rotations for recursive least squares, PhD Thesis, University of Warwick.

Wiltgen T (2007) Adaptive Beamforming using ICA for Target Identification in Noisy Environments. MSc Thesis, Virginia Polytechnic Institute and State University.

13

Low Power FPGA Implementation

13.1 Introduction

As has been indicated in the introduction to this book, a lot has been made about improving technology and how this has offered increased numbers of transistors, along with improving the speed of the individual transistors. This has been central to the growth of FPGA technology and largely responsible for the change in constitution of FPGA technology during the emergence of platform SoC devices. This technological evolution has been largely responsible for driving the success of FPGA technology as the programmability aspect allows the user to develop circuit architectures with a high levels of parallelism and pipelining that are ideally suited to DSP applications. This degree of control allows the user to produce high performance without the need to resort to high clock rates as in processor implementations.

One issue that deserves special attention is power consumption. Power consumption scales down with technology evolution, which would suggest that it should be less of an issue. The problem, though, is that the number of transistors that can be associated with a single implementation has also increased, thereby increasing the power consumed by a single chip. However, this represents only one aspect of the problem. The major issue is that the nature of how power is consumed in silicon chips is changing as technology evolves, and this has severe implications for FPGAs. One trend is that the leakage power which was treated as negligible many years ago and virtually ignored, is now becoming dominant; this is a problem for all microelectronics devices. However, the power needed to charge and discharge the interconnect in chips as compared with that required by the transistors, which had been an increasingly important problem over the past decade, is particularly relevant to FPGA implementation due to the programmable interconnection provided. This is known as *dynamic* power consumption and worsens as technology evolves. It will therefore mean that interconnect will play a much more dominant part in FPGA implementations when compared with equivalent designs in ASIC technology. Of course, the programmable nature of FPGA means that we can act to reduce this power consumption, which is not the case for processors where the underlying architecture developed for ease of programmability is particularly unsuited for low power implementations.

There are a number of important reasons therefore, for reducing power consumption. Increased power consumption leads to an increase in device temperature which if uncontrolled can have severe implications for device reliability. Lower power consumption has the impact of reducing heat dissipation leading to lower costs for thermal management, both in terms of packaging costs to dispense the heat from the chip die thereby avoiding the ghastly aluminium towers seen on many microprocessor chips, and also reducing the cost or even reducing the need for fan-based cooling systems usually needed to ensure better airflow for lower temperatures of the complete board. Work

© 2008 John Wiley & Sons, Ltd

(Xilinx Inc. 2007) indicates that 'a decrease of 10°C in device operating temperature can translate to a 2× increase in component life'. A lower-power FFGA implementation has immediate benefits to the design of the power supply of the complete system, which could result in fewer and cheaper components. For example, the implementation cost for a high-performance power system has been estimated as being between US$0.50 and US$1.00 per watt (Xilinx Inc. 2007). Thus, reducing the power consumption of the FPGA implementation has clear cost and reliability impact.

With battery technology not really offering any major innovations to increase energy without increasing system cost or weight, portability is now relying heavily on reducing device power consumption. As new services are introduced in mobile hardware, this translates to an increasing system performance budget, which must be achieved within the existing, or more realistically, a reduced power budget. To reduce risk, this would imply using programmable hardware solutions which immediately implies high power consumption.

The purpose of the chapter is to give an overview of the sources of power consumption and the design techniques that can be used to reduce it. It starts off in Section 13.2 by looking at the various sources of power and describes how they will be vary in the future by referring to the International Technology Roadmap for Semiconductors (IRTS 2003). Section 13.3 briefly reviews the possible approaches to reduce power consumption. Voltage reduction is introduced in Section 13.4, as a technique to reduce static power consumption. However, given the limited scope to reduce power through voltage scaling in FPGA due to the fact that the voltage has been carefully chosen for safe operation of the device, a number of techniques for reducing dynamic power consumption are then outlined in Section 13.5. These include use of pipelining, imposition of locality, data reordering and fixed coefficient exploitation. In Section 13.10, some of these techniques are then applied to the design of an FFT and real power results, presented.

13.2 Sources of Power Consumption

Power consumption in a CMOS technology can be defined in two parts, namely *static* power consumption which as the name suggests is the power consumed when the circuit is in static mode, switched on, but necessarily processing data, and *dynamic* power consumption which is when the chip is processing. The static power consumption is important for battery life in standby mode as it represents the power consumed when the device is powered up and dynamic power is important for battery life when operating as it represents the power consumed when processing data. The static power consumption comprises a number of components, as shown in Figure 13.1.

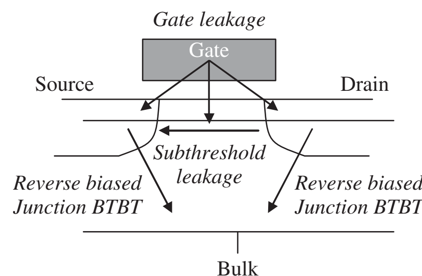


Figure 13.1 Sources of leakage components in MOS transistor

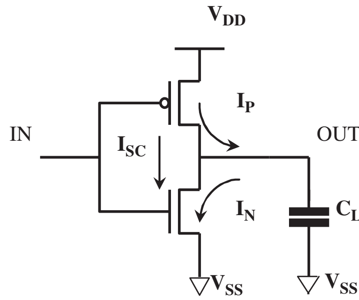


Figure 13.2 Simple CMOS inverter

The *gate leakage* is current that flows from gate to substrate, *source-to-drain leakage* (known as the *subthreshold current*) is the current that flows in the channel from the drain to source, even though the device is deemed to be off (i.e. the gate-to-source voltage V_{GS} is less than the threshold voltage of the transistor V_{T}) and *reverse-biased BTBT currents* which are the currents that flow through the source–substrate and drain–substrate junctions of the transistors when the source and drain are at higher potential than the substrate.

13.2.1 Dynamic Power Consumption

For dynamic power consumption, the leakage through a simple inverter, given in Figure 13.2, is considered. Assume that a pulse of data is fed into the transistor charging up and charging down the device. Power is consumed when the gates drive their output to a new value and is dependent on the resistance values of the p and n transistors in the CMOS inverter.

Thus, the current through the capacitor and the voltage across it are given as (Wolf 2004):

$$i_C(t) = \frac{V_{\mathrm{DD}} - V_{\mathrm{SS}}}{R_{\mathrm{p}}} e^{(-t/R_{\mathrm{p}}C_{\mathrm{L}})} \tag{13.1}$$

where the voltage is given by:

$$v_C(t) = V_{\mathrm{DD}} - V_{\mathrm{SS}}\left[1 - e^{(-t/R_{\mathrm{p}}C_{\mathrm{L}})}\right]. \tag{13.2}$$

giving the energy required to charge the capacitor as:

$$E_{\mathrm{C}} = \int i_{C_{\mathrm{L}}}(t)V_{C_{\mathrm{L}}}(t))\mathrm{d}t, \tag{13.3}$$

$$= \left[C_{\mathrm{L}}(V_{\mathrm{DD}} - V_{\mathrm{SS}})^2 \left(e^{-t/R\mathrm{p}CL} - \frac{1}{2}e^{-2t/R\mathrm{p}CL}\right)\right]_0^{\infty}, \tag{13.4}$$

$$= \frac{1}{2}C_L(V_{DD} - V_{SS})^2 \tag{13.5}$$

Table 13.1 Typical switching activity levels (Chandrakasan and Brodersen 1996)

Signal	Activity (α)
Clock	0.5
Random data signal	0.5
Simple logic circuits driven by random data	0.4–0.5
Finite state machines	0.08–0.18
Video signals	0.1(MSB)–0.5(LSB)
Conclusion	0.05–0.5

As the same charge will then be dissipated through the n-type transistor when the capacitance is discharging, therefore, in a cycle of operation of the transistor, the total energy consumption of the capacitance will $\frac{1}{2}C_L(V_{DD} - V_{SS})^2$. When this is factored in with the normal operation of the design, which can be assumed to synchronous operating at a clock frequency of *f*, then this will define the total power consumed, namely $\frac{1}{2}C_L(V_{DD} - V_{\mathrm{SS}})^2 f$. However, this assumes that every transistor is charging and discharging at the rate of the clock frequency which will never happen. Therefore, a figure to indicate what proportion of transistors are changing, namely α, is introduced. For different circuits, the value of α will vary, as shown in Table 13.1 (Chandrakasan and Brodersen 1996).

This gives the expression for the dynamic power consumption of a circuit, P_{dyn} as shown in Equation (13.6):

$$P_{\mathrm{dyn}} = \frac{1}{2}C_{\mathrm{L}}(V_{\mathrm{DD}} - V_{\mathrm{SS}})^2 f\alpha \tag{13.6}$$

which, when V_{SS} is assumed to be 0, reduces to the better-known expression of Equation (13.7):

$$P_{\mathrm{dyn}} = \frac{1}{2}C_{\mathrm{L}}V_{\mathrm{DD}}^2 f\alpha \tag{13.7}$$

In addition, short-circuit current can be classified as dynamic power consumption. Short-circuit currents occur when the rise/fall time at the input of a gate is larger than the output rise/fall time, causing imbalance and meaning that the supply voltage V_{DD} is short-circuited for a very short space of time. This will particularly happen when the transistor is driving a heavy capacitative load which it could be argued, can be avoided in good design. To some extent therefore, short-circuit power consumption is manageable.

13.2.2 Static Power Consumption

The scaling of technology has provided the impetus for many product evolutions as it gives a scaling of the transistor dimensions, as illustrated in Figure 13.3. Simply speaking, scaling by k means that the new dimensions shown in Figure 13.3(b) are given by $L' = 1/k(L)$, $W' = 1/k(W)$ and $t_{\mathrm{ox}} = 1/k(t_{\mathrm{ox}})$. It is clear that this will translates to an k^2 increase in the number of transistors and also an increase in transistor speed; an expected decrease in transistor power as currents will also be reduced.

The expected decrease in power consumption, however, does not transpire. In order to avoid excessively high electric fields in the scaled structure, it is necessary to scale the supply voltage

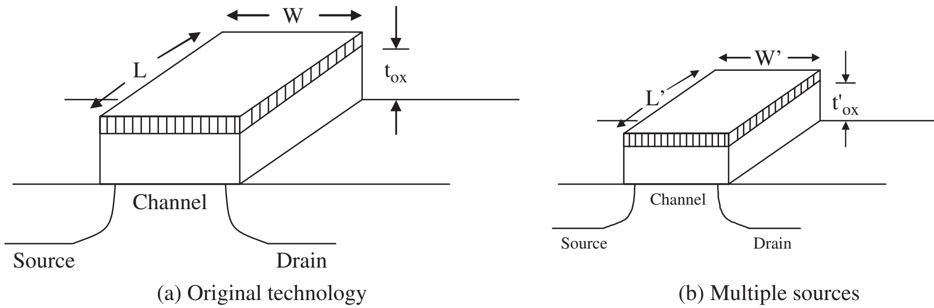


Figure 13.3 Impact of transistor scaling

V_{DD}. This in turn requires a scaling in the threshold voltage V_{th}, otherwise the transistors will not turn off properly. The reduction in threshold voltage results in an increase in subthreshold current. In order to cope with the short channel effects which have come about as a result of scaling the transistor dimensions, it is typical to scale the oxide thickness which results in high tunnelling through the gate insulator of the transistor, leading to the gate leakage. This gate leakage is therefore inversely proportional to the gate oxide which will continue to decrease for improving technologies, therefore exacerbating the problem.

Scaled devices also require high substrate doping densities to be used near the source–substrate and drain–substrate junctions in order to reduce the depletion region. However, under high reversed bias, this results in significantly large BTBT currents through these junctions (Roy *et al*. 2003). The result is that scaling results in a dramatic increase in each of these components of leakage and, with increasing junction temperatures, the impact is worsened as the leakage impact is increased (Xilinx Inc. 2007).

The main issue with increasing number of transistors is that their contribution to static power consumption is also growing. This is illustrated from the graph in Figure 13.4, taken from International Technology Roadmap for Semiconductors (IRTS 2003), (Kim *et al*. 2003) which shows that a cross-over point is emerging for 90 nm and smaller technology nodes where static power will begin to eclipse dynamic power for many applications.

This graph has a major impact for many technologies as it now means that, unlike power consumption in the previous decade where the power consumption issue largely impacted the dynamic operation of the device, allowing designers to reduce its impact, the power consumption will not be predicated on the normal standby mode of operation for the device. This will have impact for system design for fixed, but particularly for wireless applications. A number of approaches are being adopted to address this.

Xilinx has attempted to address the impact of high static power in the Virtex-4 and following devices by employing a triple-oxide in their 90 nm FPGA (Xilinx Inc. 2007). *Triple-oxide* is used to represent the three levels of oxide thickness used in FPGAs. A *thin-oxide* is used for the small, fast, transistors in the FPGA core, a *thick-oxide* is used for the higher-voltage, swing transistors for the I/O which do not have to be fast, and a third-level oxide called a *middle-thickness oxide* is used for the configuration memory cells and interconnect pass transistors. These transistors operate at a higher threshold voltage than the thin-oxide transistors, but as they are only used to store the configuration for logic and data passing, there is no strict requirement for speed. This represents a sensible trade-off in reducing power consumption and is not really applicable to technologies other than FPGAs such as microprocessors and DSP processors, unless they have transistors for storing configuration data.

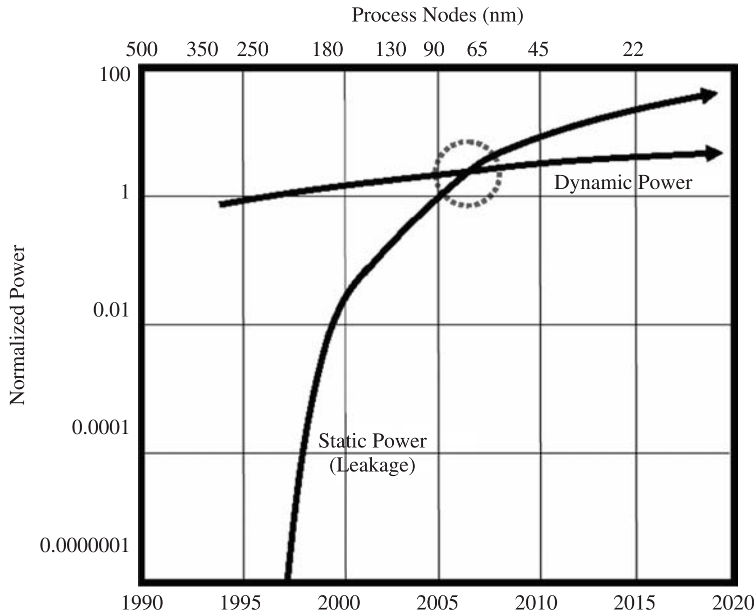


Figure 13.4 Impact of static versus dynamic power consumption with technology evolution (Kim *et al*. 2003). Reproduced from © IEEE

Altera employ a similar strategy using two types of transistors, namely a transistor with low-voltage threshold and small minimal channel length for high-speed operation in the DSP blocks and logic elements, and a low power transistor with a higher threshold voltage and larger channel length to implement the less demanding areas of the chip in terms of performance such as the configuration RAM and memory blocks.

Whilst addressing static power consumption is therefore vital in developing a low power FPGA solution, it is somewhat determined by the underlying application of many of the transistors used in FPGA architectures; some of these used for storage and control, and are therefore not under the direct influence of the designer. In addition, the device will have already been optimized for low power performance as outlined above. There are however, some techniques that can act to reduce the power consumption from a static power consumption perspective.

Clock Tree Isolation

One of the main contributors to static power consumption is the clock signal through its distribution network, namely a clock tree and the circuits connected to it which it will act to toggle on a regular basis, particularly if the design is synchronous. As the description in Chapter 5 indicated, most FPGAs have a number of clock signals with PPLs and individual dedicated clock tree networks which can be turned off and on as required. This is done by using clock multiplexing to turn off parts of the FPGA.

In the Virtex-4 FPGA, this is achieved using a clock gating block which provides an efficient means of turning off a global clock net. This has much more impact than simply disabling flip-flops as it allows the large toggling net to be turned off, thereby avoiding the dynamic power dissipation due to the intense toggling. Moreover, it should be noted that unlike Equation (13.7), these nets are being clocked at the clock frequency and hence, will produce a much higher power reduction.

13.3 Power Consumption Reduction Techniques

By examining the expression of Equation (13.7), it is clear that there are number of factors that can be varied for dynamic power consumption. The power supply voltage V_{DD} will obviously have been predetermined by the FPGA vendor (and optimized to provide low power operation) and any scope to reduce this voltage will be made available to the user via the design software. This only leaves adjustment of the other parameters, namely the toggling rate which will be determined by the clock frequency f, the switching activity α, and the load capacitance C_L. However, any technique that acts to adjust the clock frequency f and/or switching activity α should be developed on the understanding that the overall clock rate for the system will have generally been determined by the application, and that the switching activity will be governed again by the application domain, meaning that levels shown in Table 13.1 should be given consideration.

Generally speaking, power reduction techniques either act to minimize the switched capacitance Cf or employ techniques which reduce the supply voltage by increasing the system's throughput beyond that necessary, either through the use of parallel hardware, thereby increasing area and therefore capacitance, or increasing the speed (Chandrakasan and Brodersen 1996). The voltage is then reduced, slowing up performance until the required throughput rate is met, but at a lower power consumption budget. This latter approach acts to give a squared reduction in power, at the expense of a linear increase in area, i.e. C_L and/or frequency f. This is a little more difficult in FPGAs, but work by (Chow *et al*. 2005) suggests appropriate techniques and will be described in Section 13.4.

There is also some scope to reduce the capacitance and the switching activity, but rather than consider this separately, it is useful to think about reducing the *switched capacitance* of a circuit, i.e. the sum of all of toggling activity of each node multiplied by the capacitance of that node. This is an important measure of power consumption as opposed to just circuit capacitance alone, as a circuit can either have a 'large'capacitative net with a 'low' switching activity which will not contribute greatly to power consumption, or a number of 'low' capacitance nets with a 'lot'of switching activity which can contribute to power consumption. The same argument applies to switching activity levels; some nets can have high switching activity, but a low capacitance and so on. A large proportion of the techniques fall into this domain, so detailed consideration is given in Section 13.5.

13.4 Voltage Scaling in FPGAs

As the name indicates, voltage scaling involves reducing the supply voltage of the circuit in such a way that the circuit can still operate correctly. Typically, the designer will have exploited any voltage capacity by applying design techniques to slow down the circuit operation, presumably by achieving an area reduction or some other gain. Reductions in the voltage may cause a circuit failure as the critical path timing may or may not, be met. This is because scaling the voltage causes affects the circuit delay, t_d, as determined by the expression below (Bowman *et al*. 1999):

$$t_d = \frac{kV_{DD}}{(V_{DD} - V_t)^2} \tag{13.8}$$

where k and α are constants with $1 < \alpha < 2$. As V_{DD} is scaled, the circuit delay t_d increases.

The impact of voltage scaling can be addressed in two ways, adding circuitry to detect exactly when this happens with a specific FPGA implementation and detecting the correct voltage threshold to achieve lower-power operation (Chow *et al*. 2005), or applying design techniques to speed up the circuit operation.

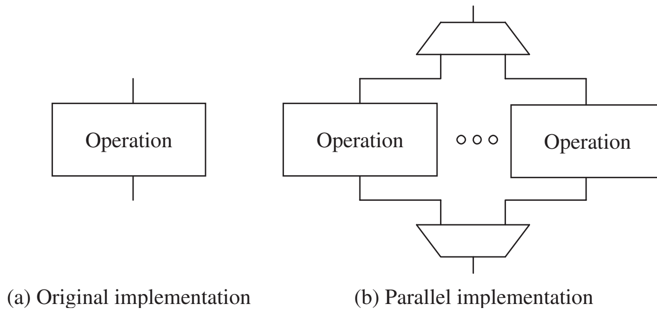


Figure 13.5 Use of parallelism (and voltage scaling) to lower power consumption

One such approach is to speed up the computation, as shown in Figure 13.5. If we presume the operation shown in Figure 13.5(a) matches the speed requirements at the operating voltage, then if parallelism can either be added to the circuit implementation or possibly exploited if it naturally exists (as is the case in many DSP systems), then the circuit shown in Figure 13.5(b) can result. This resulting circuit now operates much faster than the original circuit, so it is possible to apply voltage scaling; this will result in a power reduction due to the V_{DD}^2 voltage scaling as well as the scaling in the frequency. Of course, the circuit in Figure 13.5(b) will have a larger area and, therefore, capacitance and switching activity, but the squared reduction and frequency reduction will more than outweigh this gain.

It is also important to consider the possible means by which voltage scaling can be applied to FPGAs. This should only applied to the FPGA core as the device will be used in a wide range of applications; this is possible as there are many V_{DD} pins supplied to the core and I/O ring. It is important that the I/O pins operates to the specifications to which they have been designed, particularly as some static power consumption techniques will have already been applied to some FPGA devices such as the Xilinx Virtex-4 and -5 families and Altera's Stratix II and III FPGA families. Reducing the voltage has the impact of increasing the circuit delay, but given that only parts of the circuit need to operate at the slowest circuit delay, there is scope for reducing the voltage for a large portion of the circuit without impacting performance. Of course, the design has to be reliable across a range of devices and there can be a variation in delay times as well as operating temperature.

The work in Chow *et al.* (2005) follows the first principle of assuming the original circuit, altering the internal voltage supply and then checking for any possible errors (although it can of course be used for the second approach shown in Figure 13.5). They argue that on the basis that the designer can observe any two types of design errors as a result of the voltage scaling in normal operation, then it is just a case of trying to work out two other types of error: *I/O errors* which have resulted as the lower-voltage core circuit has to interface to the I/O which is operating at the original voltage; and *delay errors* which occur as a result of the critical path now possibly not meeting the timing requirement. In the case of *I/O errors*, the danger is that a high output signal from the core will be too small for the threshold voltage of the I/O buffer to correctly detect it, as a high value; this means it will incorrectly interpret it as a low value.

Chow *et al.* (2005) use a logic delay measurement circuit (LDMC) (Gonzalez *et al.* 1997) which is used in addition to the designer's FPGA circuit and which consists of a delay line or inverter chain, a chain of flip-flops, and a leading zero detector implemented using normal FPGA logic resources. The circuit effectively generates a warning signal by propagating a wave front through the inverter chain using the clock signal, and then measuring how many of the flip-flops will have correctly

clocked the data using the same clock signal. Thus, the number of inputs that have switched will depend on the delay of the inverters which in turn will depend on temperature and supply voltage; the circuit's propagation delay can then be computed using this leading-zero detector. In this way, the LDMC measures how many delay stages that the falling edge propagates in half of a clock period. The circuit is investigated for a number of circuits and the authors show power reductions in the range of 4–54%, although they would typically expect to achieve 20–30% which is still impressive. The main limitation of this approach is that the design would have to be investigated for each FPGA operating in its specific environment as circuit performance will vary from FPGA to FPGA. Given the relationship between power consumption, delay and temperature, it cannot be assumed that the performance will be the same from device to device. This is avoided in normal FPGA operation by specifying for worst-case operation. However, they are a number of other low power design techniques that do not require alteration of FPGA parameters that the designer may want to investigate before adopting such an approach.

13.5 Reduction in Switched Capacitance

The previous techniques require that the voltage is scaled (typically only the internal voltage), but does not deal with the results in the application of this scaling. However, as was suggested in Section 13.3, another technique to reduce power consumption involves reducing the switched capacitance of the circuit. As demonstrated in Equation (13.7), this produces a squared ($f \times C_L$ product) reduction in dynamic power consumption. The resulting FPGA layout can have a major influence on the switched capacitance as the layout determines first, the capacitance of the routed interconnect and therefore to some extent, the switching activity due to the different delays in various routes (although overall switching will be governed by input data and circuit functionality). However, the FPGA layout can be heavily influenced by the architectural style developed to create the design, meaning that, as will be seen in this section and in the FFT example (Section 13.10), architectural decisions can have a major impact on power consumption.

A number of techniques are considered which are well understood in the literature. These techniques are not covered in detail and represent only a number of possible optimizations possible.

13.6 Data Reordering

In DSP processor implementations described in Chapter 4, the architecture is typically composed of data and program memory connected to the processor via data busses. In these architectures therefore, the capacitance of the buses will be fixed, but in some cases it may be possible to reorder the data computation in order to minimize the Hamming difference and thereby achieve a reduction in the switching activity on the large capacitative buses. Consider a simple 4-tap filter with 4-bit coefficients listed below:

```
a0  1011          a0  1011
        |3                |1
a1  0110          a2  1001
        |4                |3
a2  1001          a3  0100
        |3                |1
a3  0100          a1  0110
        |4                |3
a0  1011          a0  1011
8 transitions     3 transitions
```

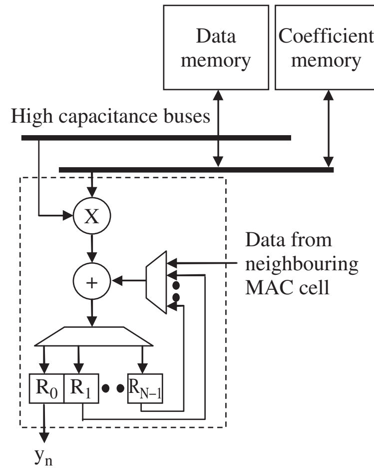


Figure 13.6 Generic MAC time domain filter implementation (Erdogan and Arslan 2002)

It can be seen that if the coefficients are loaded in the normal numerical order, namely a_0, a_1, a_2, a_3 and back to a_0, then this will require 14 transitions. This will involve charging and discharging of the line capacitance of the interconnect to load these coefficients into the processing engine. Depending on the architectural decisions made, this interconnection length could be considerable. For example, a DSPμ-style implementation was chosen where data is loaded from coefficient memory across a lengthy bus, so this would translate to quite a large contribution to switched capacitance. By changing the order of loading, the number of transitions can be reduced to 8 as shown. The main issue is then to resolve the *out-of-order* operation of the coefficient. Whilst this may be an issue in some circuit architecture implementations where the selection of the coefficients is done using *fixed* multiplexers which cannot be changed and which will never be the case for DSP processor implementations. This works well in applications where the coefficients are predetermined and will not change. If this was the case, it would be possible to develop the circuit architecture using many of the techniques outlined in Chapter 8, to produce an architecture where this was exploited. For example, hardware sharing could be controlled based on the switching activity of the coefficients.

In (Erdogan and Arslan 2002), the authors show how this can be applied to the design of FIR filters to achieve a reported 62% reduction in power consumption. The architecture shown in Figure 13.6 has been reproduced from their paper and represents a MAC structure comprising a multiplier and adder which are fed by program and coefficient data from the memory. The structure presented can be made to be cascadable by feeding the output of the previous section into the current block. This structure clearly supports *out-of-order* operation to be performed, resulting in a reduction of the switching data from the coefficient memory via the large coefficient data bus. By exploiting the direct form FIR filter implementation rather than the transposed form, this also reduces the switching on the data bus as one data word is loaded and then reused.

13.7 Fixed Coefficient Operation

The DA and RCM techniques presented in Chapter 6 represent an optimization that can be applied in addition to the reordering technique. These techniques were originally presented as they allowed both a reduction in area and in some cases, an increase in speed. However, the underlying concept

for both approaches is that they reduce the amount of hardware required to perform the computation of a full multiplication, and therefore by implication, reduce the amount of interconnection needed to construct the multipliers and therefore the switched capacitance accordingly. In the case of the RCM, only part of the multiplier structure is implemented to perform the computation, so scope exists to create additional multiplexing features which could be used to turn off parts of the multipliers, although this had not been investigated in detail.

13.8 Pipelining

An effective method to reduce power consumption is by using pipelining coupled with power-aware component placement. Pipelining, as illustrated in Figure 13.7, breaks the processing of the original circuit of Figure 13.7(a) into short stages, as illustrated in Figure 13.7(b), thereby providing a speed-up, but with increasing the latency in terms of the number of clock cycles, not necessarily the actual time (as the clock period has been shortened). The increase in processing speed can be used in a similar way to the use of parallelism in Figure 13.5, to allow the voltage to be reduced, thereby achieving a power reduction (Chandrakasan and Brodersen 1996). Indeed as seen in earlier chapters, both techniques can be applied to provide considerable speed improvement.

In addition to providing the speed-up, pipelining provides a highly useful mechanism to reduce power consumption (Raghunathan 1999, Keane *et al.* 1999), particularly in FPGA (Wilton *et al.* 2004). In FPGA designs, the aim of the place and route tools is to achieve the best placement in order to achieve the required speed (and more recently, power-efficient realization). Pipelining provides more rigorous structure to the design, allowing faster placement and also reduces the number of longer nets that result in the design. Moreover, the pipelined circuit suffers fewer glitches than an unpipelined version, since it typically has fewer logic levels between registers. Thus, there is a reduction in the overall and average netlengths and a decrease in the switching activity, as a highly pipelined circuit suffers fewer glitches than an unpipelined circuit, since it typically has fewer logic levels between registers, which results in less dynamic power being consumed.

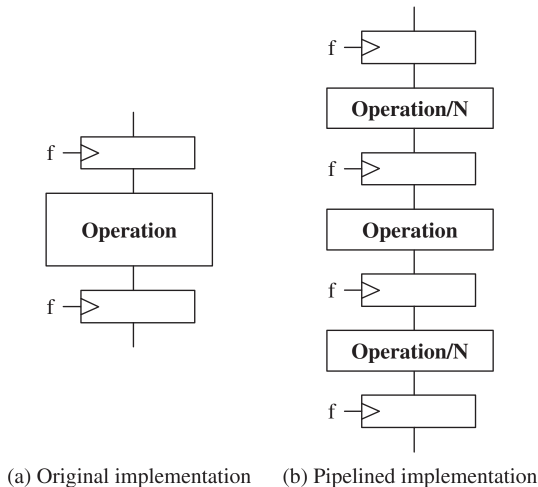


Figure 13.7 Application of pipelining

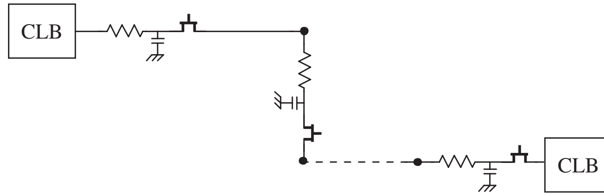


Figure 13.8 Typical FPGA interconnection route

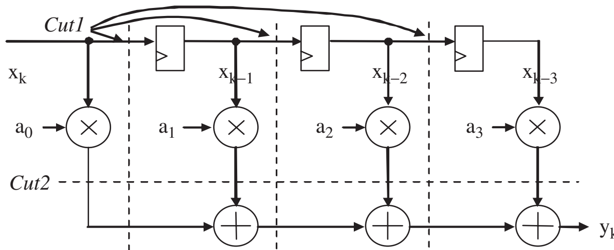


Figure 13.9 Pipelined FIR filter implementation for low power

This technique is particularly effective in FPGA technology because the increasing flexibility comes at a power budget cost due to the long routing tracks and programmable switches. These features afford the programmability, but are laden with parasitic capacitance (Chen *et al*. 1997), as illustrated by Figure 13.8 which shows a model of a typical FPGA route. Another benefit of implementing pipelining in a FPGA is that it may be using an under-utilized resource, namely the flip-flop at the output of the logic cell, thereby only providing a small area increase (Wilton *et al*. 2004). Whilst it could be argued that the flip-flop would not have been used in the unpipelined version and is therefore not contributing to power, it should be noted that it will still probably have a clock supplied to it which will act to dominate the power consumption.

Consider the application of pipelining to a FIR filter given in Figure 13.9. The filter is a simple delay line filter structure as described in earlier chapters. Two levels of pipelining will be investigated, namely a layer of pipelining after the multipliers, as shown by *Cut2* and given as PL1 in Table 13.2, and another pipeline cut in the adder (and delay) chain, *Cut1* in addition to the first level of pipelining, given as PL2 in Table 13.2, i.e. *PL2* encompasses *Cut1 and Cut2*.

A number of filter realizations were investigated namely a 4, 8, 16, and 32 tap FIR filter implemented on a Virtex-II XC-2V3000bf 957-6 FPGA (McKeown *et al*. 2006). The filter is initialized by loading coefficients using an address bus, data bus and enable signal so that this was consistent for all implementations. No truncation was employed and the output word length is 24 and 27 bits for the 4-tap and 32-tap filters respectively. Expansion is handled by including a word growth variable which is defined for each filter size to prevent truncation. The unpipelined version (PL0) is used to benchmark the power consumption before reduction techniques are applied. Functionality remains the same in all versions, except for the latency imposed by pipelining stages. A speech data file is used to simulate the designs, filter clock speed is 20 MHz and simulation time is 200 µs.

The FIR filters were coded in VHDL with generic parameters used for coefficient and input data word sizes. Three filters (band-pass, low-pass and high-pass) were created in Matlab® for 4, 8, 16 and 32 tap filters and filter coefficients extracted and used as input. ModelSim™ and Synplicity®

Synplify Pro® were used for simulation and synthesis respectively. Xilinx ISE™ Project Navigator (version 6.2) was used to translate, map, place and route the designs and sub-programs of the ISE™ design suite were used to compile component libraries, manually place and route and generate post place and route VHDL files for XPower. Xpower was then used to generate simulation results for Table 13.2. The flow is given in Figure 13.10. The results shows power reductions of 63–59% for

Table 13.2 Internal signal/logic power consumption of various FIR filters (McKeown *et al.* 2006)

	Filter tap size			
Technique	4	8	16	32
PL0	8.4	89.7	272.0	964.2
PL1	3.1 (−63%)	29.1 (−68%)	89.7 (−67%)	391.7 (−59.3%)
PL2	1.5 (−82%)	6.7 (−93%)	8.1 (−97%)	16.4 (−98%)

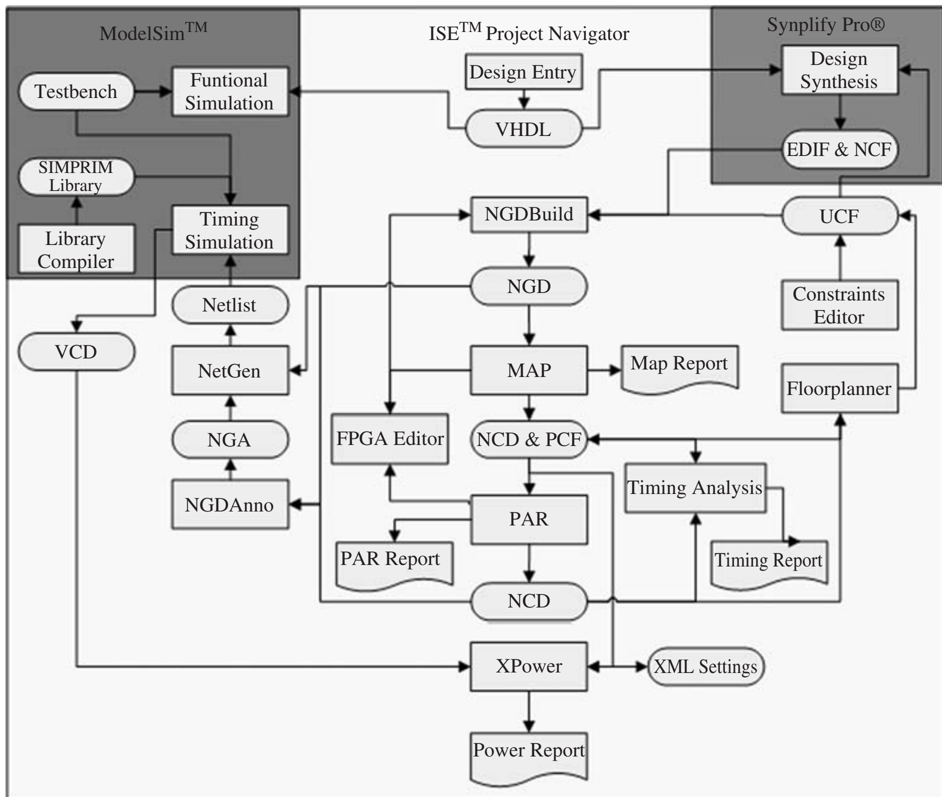


Figure 13.10 Detailed design flow for Xpower estimation

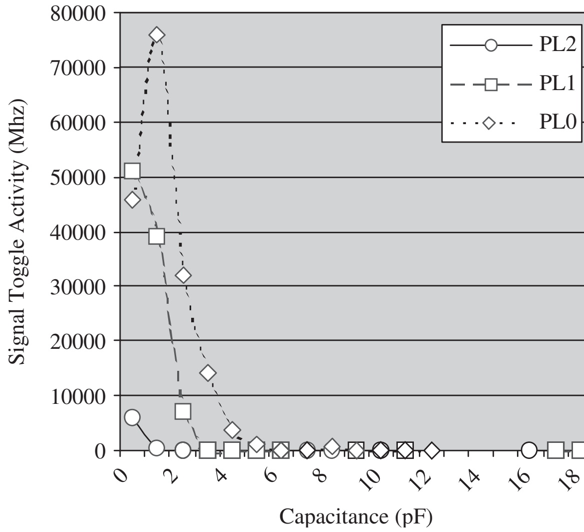


Figure 13.11 Tap signal capacitance and toggle activity

single stage pipelined versions and 82–98% for the two stage pipelined versions, depending on filter size.

It is clear from the table that the pipelining becomes more effective as filter size and thus design area and interconnection lengths increase; this gives greater opportunity for power-aware placement. Net lengths are shortened by placing interconnected components closely together. This reduces power consumption in two ways, first by decreasing the net capacitances and second by reducing unequal propagation delays. Partitioning the design into pipelined stages further reduces power consumption by diminishing the ripple effect of propagation delays. This can be seen in Figure 13.11 which shows post place and route capacitances rounded to the nearest integer, plotted against the summation of the toggling activity on nets with equivalent capacitances. These values are plotted for PL0, PL1 and PL2 versions and show that, not only is toggle activity reduced on high capacity nets, but overall there are fewer toggles in the design when power reduction techniques are implemented.

The results presented in Wilton *et al*. (2004) are more thorough as they are based on data from a real board set-up and are equally impressive. They have presented results for a 64-bit unsigned integer array multiplier, a triple-DES encryption circuit, an 8-tap floating point FIR filter and a Cordic circuit to compute sine and cosine of an angle. A summary of the power results just for the FIR filter and the CORDIC circuit are given in the Table 13.3. They were taken from the circuits implemented on an Altera Nios Development Kit (Stratix Professional Edition) which contains a 0.13 µm CMOS Stratix EP1S40F780C5 device; this was used to produce the original FPGA power results and the estimated power was taken from Quartus simulator and power estimator.

The results show the impact of applying of different levels of pipelining. They quote savings overall of 40–82% and indicate that, when quisicient power is factored out from the results, the savings on the dynamic logic block energy can be as high as 98%. They indicate that lower-level physical

Table 13.3 Pipelining results for 0.13 μm FPGA

Benchmark circuit	Pipeline stages	Estimated power	Original FPGA power
8-tap floating point FIR filter	2	4420	7866
	4	2468	5580
	Max.	776	3834
Cordic circuit to compute sine and cosine of angle	4	971	5139
	8	611	4437
	16	565	4716
	Max.	567	4140

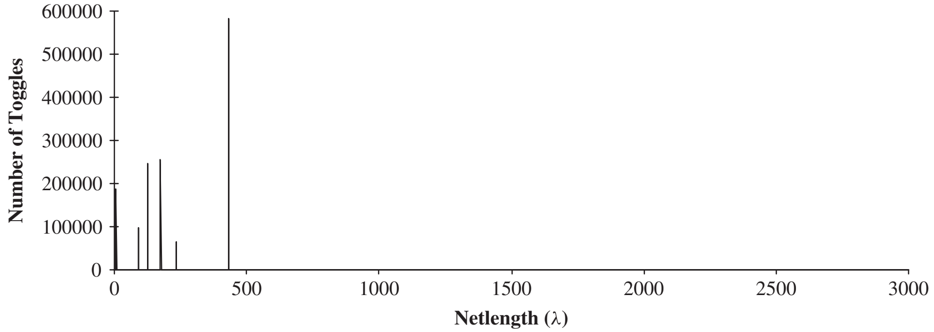
design optimizations presented in the work of (Lamoureux and Wilton 2003) can only achieve energy savings of up 23%, highlighting the importance of applying system-level optimizations, and highlighting the impact of pipelining generally.

13.9 Locality

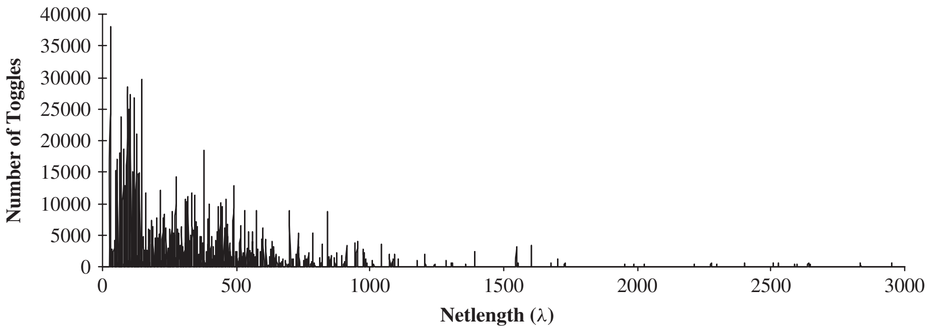
As discussed in Chapter 4, systolic arrays (Kung and Leiserson 1979, Kung 1982) were initially introduced as a solution to solve the design problems inherent at the time, namely design complexity and the increasing problem of long interconnect in VLSI designs (Mead and Conway 1979). Systolic array architectures have many attractive features, including data and processor regularity, as well as locality in terms of processor interconnection. The early structures were developed for regular computations such as matrix–matrix multiplication and LU decomposition which involved a combination of parallel and pipelined processing. Systolic array architectures take advantage of the highly regular nature of DSP computations and offered huge performance potential.

The key attraction of systolic array architectures is that they provide an architectural framework to ensure regularity in the development of FPGA circuit architectures. As was demonstrated in the pipelining example in the previous section, an architectural optimization can bring benefits in terms of reduced power consumption. The work of Keane *et al*. (1999), showed clearly how creating architectures, in this case bit-level architectures for multipliers, could provide reduced power consumption. The comparison was based on ASIC, but the key message from the paper was the importance of preserving locality. Figure 13.12 shows the switching activity for different netlengths for a carry-save multiplier (Figure 3.6) and a Wallace tree multiplier (Figure 3.7).

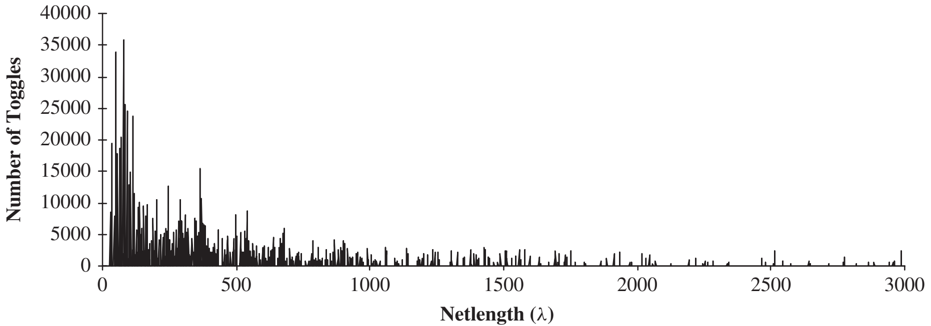
The first graph Figure 13.12(a) shows how the switching is limited to a few netlist length sizes when regularity is preserved in the layout process. The second graph Figure 13.12(b) shows the impact when the same design is flattened during synthesis for the same structure. The final graph Figure 13.12(c) gives the distribution for the Wallace tree. It should be noted that higher activity occurs on the larger nets. Both designs have similar transistor counts and toggling activity for the simulation undertaken, and clearly indicate that this effect is responsible for the 40% increase in power in the Wallace tree implementation. The problem worsens as wordlength grows. The application of locality to FPGA implementations is demonstrated next in the application of the approach to an FFT implementation.



(a) Carry-save with regularity preserved



(b) Carry-save multiplier based on flattened netlist



(c) Wallace tree multiplier based on flattened netlist

Figure 13.12 Switched capacitance activity for different multiplier structures

13.10 Application to an FFT Implementation

Many of these concepts can be applied to a real DSP example, in this case an FFT design implementation which can be implemented in many ways. There are a number of texts that look at FPGA implementation Meyer-Baese (2001). Consider the discrete Fourier transform (DFT) of N complex

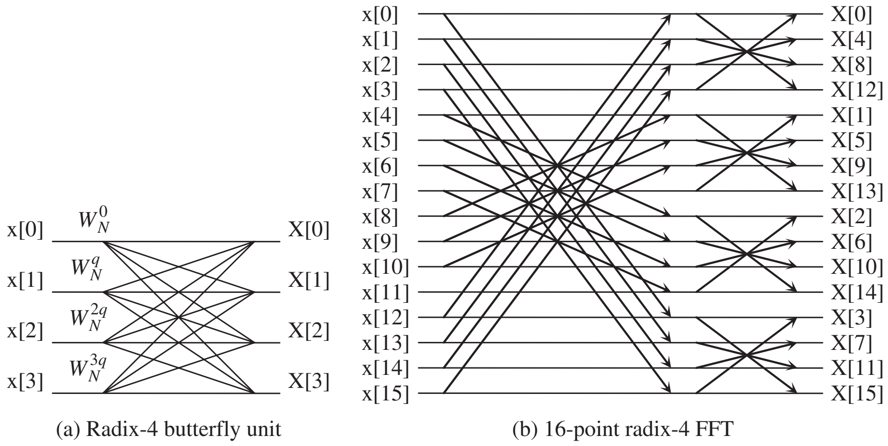


Figure 13.13 FFT structure

data points, $x(n)$is defined by:

$$X_k = \sum_{n=0}^{N-1} x(n) W_N^{nk} \qquad k = 0, 1, \ldots, N-1 \tag{13.9}$$

where W_N is the twiddle factor $W_N = e^{-j(2\pi/N)}$. The FFT can be derived from this, as presented in Bi and Jones (1989) which involves rewriting the DFT expression. By far the most common and widely used of these is the Cooley–Tukey algorithm (Cooley and Tukey 1965) which reduces the algorithmic complexity from $O(N^2)$ to $O(N \log N)$ through use of an index mapping scheme using the symmetry and periodicity in the transform coefficients. The radix-4 implementation recursively decomposes the algorithm until only 4-point DFTs are required. The results are then combined to compute the N-point transform. The FFT is computed using the butterfly unit (Figure 13.13(a)) and the perfect shuffle network (Figure 13.13(b)). The globally recursive nature of the algorithm manifests itself as global interconnect, requiring irregular routing where data is routinely passed to nonadjacent processing elements (PEs, Stone 1989). The radix-4 algorithmic expression is expressed as:

$$\begin{aligned} X_k = {} & \sum_{n=0}^{N/4-1} x(n) W_N^{nk} + W_N^{Nk/4} \sum_{n=0}^{N/4-1} x(n+N/4) W_N^{nk} \\ & + W_N^{Nk/2} \sum_{n=0}^{N/4-1} x(n+N/2) W_N^{nk} + W_N^{3Nk/4} \sum_{n=0}^{N/4-1} x(n+3N/4) W_N^{nk} \end{aligned} \tag{13.10}$$

Typically, larger point FFTs are created from small FFT blocks, but the inherent irregularity can impact speed performance and contribute to consumed power. An alternative decomposition of the computation is possible which involves identifying repetitive patterns in the structure of the

16-point DFT matrix shown in Equation (13.11).

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \\ X_8 \\ X_9 \\ X_{10} \\ X_{11} \\ X_{12} \\ X_{13} \\ X_{14} \\ X_{15} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & W & W^2 & W^3 & -j & W^5 & W^6 & W^7 & -1 & -W & -W^2 & -W^3 & j & -W^5 & -W^6 & -W^7 \\ 1 & W^2 & -j & W^6 & -1 & -W^2 & j & -W^6 & 1 & W^2 & -j & W^6 & -1 & -W^2 & j & -W^6 \\ 1 & W^3 & W^6 & -W & j & -W^7 & W^2 & W^5 & -1 & -W^3 & -W^6 & W & -j & W^7 & -W^2 & -W^5 \\ 1 & -j & -1 & j & 1 & -j & -1 & J & 1 & -j & -1 & j & 1 & -j & -1 & j \\ 1 & W^5 & -W^2 & -W^7 & -j & -W & -W^6 & W^3 & -1 & -W^5 & W^2 & W^7 & j & W & W^6 & -W^3 \\ 1 & W^6 & j & W^2 & -1 & -W^6 & -j & -W^2 & 1 & W^6 & j & W^2 & -1 & -W^6 & -j & -W^2 \\ 1 & W^7 & -W^6 & W^5 & j & W^3 & -W^2 & W & -1 & -W^7 & W^6 & -W^5 & -j & -W^3 & W^2 & -W \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -W & W^2 & -W^3 & -j & -W^5 & W^6 & -W^7 & -1 & W & -W^2 & W^3 & j & W^5 & -W^6 & W^7 \\ 1 & -W^2 & -j & -W^6 & -1 & W^2 & j & W^6 & 1 & -W^2 & -j & -W^6 & -1 & W^2 & j & W^6 \\ 1 & -W^3 & W^6 & W & j & W^7 & W^2 & -W^5 & -1 & W^3 & -W^6 & -W & -j & -W^7 & -W^2 & W^5 \\ 1 & j & -1 & -j & 1 & j & -1 & -j & 1 & j & -1 & -j & 1 & j & -1 & -j \\ 1 & -W^5 & -W^2 & W^7 & -j & W & -W^6 & -W^3 & -1 & W^5 & W^2 & -W^7 & j & -W & W^6 & W^3 \\ 1 & -W^6 & j & -W^2 & -1 & W^6 & -j & W^2 & 1 & -W^6 & j & -W^2 & -1 & W^6 & -j & W^2 \\ 1 & -W^7 & -W^6 & -W^5 & j & -W^3 & -W^2 & -W & -1 & W^7 & W^6 & W^5 & -j & W^3 & W^2 & W \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{15} \end{bmatrix} \tag{13.11}$$

Figure 13.14 shows how the roots to the left and right of the imaginary (Im) axis and the roots above and below the real (Re) axis are mirrored. The angle q of roots 1, $-j$, -1 and j follow as $2\pi/(N/4)$. Factoring out $-j$, -1 and j from the roots in the third, second and first quadrants, respectively, means that the remaining factors of these roots lie solely in the fourth 1 quadrant. If the roots are then grouped with a periodicity of $\theta = 2\pi/(N/4)$, an extremely efficient factorization of the transform matrix is obtained.

After mapping the indices and rearranging the input/output sequences of the DFT matrix in Equation (13.11), it can be partitioned into $N/4$ blocks of columns, allowing extraction of a common factor for each row, resulting in each column block being identically factorized. The transform matrix can then be partitioned row-wise into four separate ($N/4$) by ($N/4$) matrices. Further factorization along each column is then carried out resulting in the transform matrix in Equation (13.12). This implementation retains the same computational efficiencies as the Cooley–Tukey algorithm, but with increased data locality, as shown in Figure 13.15.

$$\begin{bmatrix} X(k_0, 0) \\ X(k_0, 1) \\ X(k_0, 2) \\ X(k_0, 3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix} \begin{bmatrix} a_{k1}(0) \\ W^{k1} a_{k1}(1) \\ W^{2k1} a_{k1}(2) \\ W^{3k1} a_{k1}(3) \end{bmatrix} \tag{13.12}$$

The architecture was originally developed as part of a major chip design project carried out in the 1990s, involving Queen's University Belfast, Snell and Wilcox and the BBC with the aim of

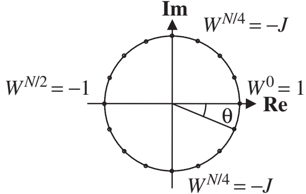


Figure 13.14 Factorization of the roots by -1, j and $-j$

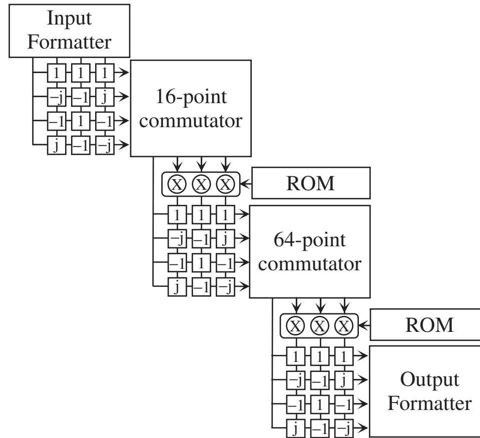


Figure 13.15 64-point FFT architecture

developing a 64-point FFT processor for use in digital TV applications (Hui *et al*. 1996). The focus of the work was to investigate if this regularity would transfer to a FPGA implementation, so a 64-point FFT (*QFFT*64) architecture was coded in VHDL and compared with *AFFT*64, a 64-point soft IP FFT core from Amphion (now Conexant) and *XFFT*64, a 64-point FFT core generated from Xilinx's Coregen IP library. The latter two designs were designed to be highly parameterized and could operate in various modes, but both would appear to be based on a similar architecture to that presented in Figure 13.13(a). Table 13.4 has been developed from results taken from the Virtex-2^{Pro} platform developed specifically to allow power comsumption measurements using a test harness and real power measurements. As the cores operated at different speeds, a number of different cores were used for each configuration namely 1 × QFFT 64, 4 × XFFT 64 and 4 × AFFT 64 cores at 800 MSPS and 3 × QFFT 64, 12 × XFFT 64 and 12 × AFFT 64 cores at 2.4 GSPS.

Figure 13.16 shows the toggling activity for the various netlengths given. As with the multiplier example shown in Figure 13.12, the regularity of the architecture has resulted in much smaller netlength, as indicated by the capacitances on the x-axis. In addition, there is dramatically reduced toggling on these nets. This represents the main saving in power consumption for the core.

In addition to the regular structure, the designs have all been pipelined, which has acted to reduce the netlengths in all of the designs. To further minimize interconnect, coefficients were pre-computed and stored in distributed RAM local to the PE. The input, output and commutator

Table 13.4 Power consumption and resource usage for 800 MSPS

	Power		Resources		
	Data set 1	Data set 2	Slices	LUTs	Mult18 × 18
QFFT64	1029	945	1950	2966	18
XFFT64	1616 (36%)	1496 (37%)	4991 (61%)	6571 (55%)	24 2(5%)
AFFT64	4044 (75%)	3288 (71%)	9697 (80%)	15871 (81%)	0
QFFT64	1029	945	1950	2966	18
XFFT64	1616 (36%)	1496 (37%)	4991 (61%)	6571 (55%)	24 2(5%)

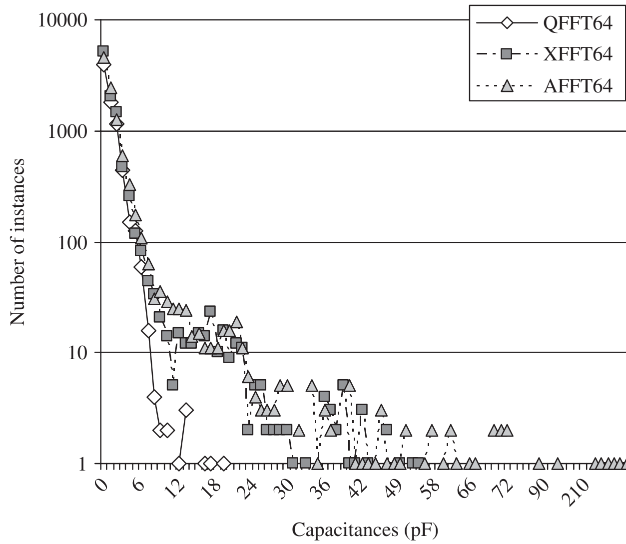


Figure 13.16 Signal capacitance for QFFT 64-point, XFFT 64-point and AFFT 64-point at 800 MSPS throughput

components consist of a series of delays and commutation multiplexers. Delay lines are implemented as shift register LUTs (SRLs) which allow variable delay length with no interconnect penalty.

13.11 Conclusions

The chapter has given a description of the sources of power consumption in FPGAs, covering both static and dynamic power consumption. Whilst some techniques were presented for static power reduction, the majority of the techniques focused on dynamic power reduction as this represented the main techniques available to the designer, due to the prefabricated nature of the FPGA. A number of techniques including data reordering, fixed coefficient functionality, use of pipelining and imposition of regularity were highlighted. These were then applied to the design of a 64-point FFT processor.

There are a number of other technqiues that merit considerations such as the development of memory architectures. One of the main advantages of FPGAs is the availability of localized memory. Typically this is not exploited as it is difficult to operate with multiple memory resources. However, developing high-level design methodologies to allow the creation of local memory in the resulting architecture is highly attractive as it represents a much more powerful, locality optimization than the one presented here. In addition, this can be driven from the dataflow level as shown in Fischaber *et al*. (2007). It is the authors' opinion that this will increasingly become an area of research over the next decade.

References

Raghunathan A, Dey S and Jha NK (1999) Register transfer level power optimization with emphasis on glitch analysis and reduction. *IEEE Trans. CAD* **18**(8), 114–131.

Bi G and Jones EV (1989) A pipelined fft processor for word-sequential data. *IEEE Transactions on Acoustic, Speech, and Signal Processing* **37**(12), 1982–1985.

Bowman K, Austin L, Eble J, Tang X and Meindl J (1999) A physical alpha-power-law mosfet model *IEEE Journal of Solid-State Circuits*, **32**, pp. 1410–1414.

Chandrakasan A and Brodersen R (1996) *Low Power Digital Design*. Kluwer.

Chen CS, Hwang TT and Liu C (1997) Low power fpga design – a re-engineering approach *Proc. 34th Design Automation Conference*, pp. 656–661.

Chow CT, Tsui LSM, Leong PHW, Luk W and Wilton S (2005) Dynamic voltage scaling for commercial fpgas *Int. Conf. on Field Programmable Technology*, pp. 215–222.

Cooley J and Tukey J (1965) An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation* **19**, 297–301.

Erdogan A and Arslan T (2002) Implementation of fir filtering structures on single multiplier dsps. *IEEE Trans. Circuits and Systems II* **49**(3), 223–229.

Fischaber S, Woods R and McAllister J (2007) Soc memory hierarchy derivation from dataflow graphs *IEEE Workshop on Signal Processing Systems*, pp. 469–474.

Gonzalez R, Gordon B and Horowitz M (1997) Supply and threshold scaling for low power cmos. *IEEE Journal of Solid-State Circuits*, **32**(8), 1210–1216.

Hui C, Ding TJ, Woods RF and McCanny JV (1996) A 64 point Fourier transform chip for video motion compensation using phase correlation. *IEEE Journal of Solid State Electronics* **31**(11), 1751–1761.

IRTS (2003) *International Technology Roadmap for Semiconductors*, 2003 edn. Semiconductor Industry Association.

Keane G, Spanier JR and Woods R (1999) Low-power design of signal processing systems using characterization of silicon ip cores. *33rd Asilomar Conference on Signals, Systems and Computers*, pp. 767–771.

Kim N, Austin T, Baauw D, Mudge T, Flautner K, Hu J, Irwin M, Kandemir M and Narayanan V (2003) Leakage current: Moore's law meets static power. *IEEE Computer* **36**(12), 68–75.

Kung HT (1982) Why systolic architectures?. *IEEE Computer* **15**(1), 37–46.

Kung HT and Leiserson CE (1979) Systolic arrays (for vlsi). *Sparse Matrix Proc. 1978*, pp. 256–282.

Lamoureux J and Wilton S (2003) On the interaction between power-aware fpga cad algorithms *Proc. Int. Conf. on Computer-aided Design*, pp. 701–708.

McKeown S, Fischaber S, Woods R, McAllister J and Malins E (2006) Low power optimisation of dsp core networks on fpga for high end signal processing systems *Proc. Int. Conf. on Military and Aerospace Programmable Logic Devices*.

Mead C and Conway L 1979 *Introduction to VLSI Systems*. Addison-Wesley Longman, Boston.

Meyer-Baese U (2001) *Digital Signal Processing with Field Programmable Gate Arrays*. Springer, Germany.

Roy K, Mukhopadhyay S and Meimand H (2003) Leakage current mechanisms and leakage reduction technqiues in deep-submicron cmos circuits. *Proc. IEEE* **91**(2), 305–327.

Stone HS (1989) Parallel processing with the perfect shuffle. *IEEE Trans. on Circuits and Systems* **36**, 610–617.

Wilton SJE, Luk W and Ang SS (2004) The impact of pipelining on energy per operation in field-programmable gate arrays *Proc. Int. Conf. on Field Programmable Logic and Application*, pp. 719–728.

Wolf W (2004) *FPGA-Based System Design*. Prentice-Hall, New Jersey.

Xilinx Inc. (2007) White paper: Virtex-5 fpgas 'power consumption in 65 nm fpgas'. Web publication downloadable from www.xilinx.com.

14

Final Statements

14.1 Introduction

The book has outlined many of the techniques needed to create FPGA solutions for DSP systems. Some of these have been encapsulated within the design tools and then applied to a number of design examples. In addition to achieving efficient FPGA implementations in terms of area, speed and throughput rates, the book has also covered the key area of low-power implementations, briefly covering a number of techniques to reduce the dynamic power consumption. The key of these approaches was to derive an efficient circuit architecture which successfully utilized the underlying resources of the FPGA to best match the computational and communication requirements of the applications. Typically, this involved using high levels of parallelism and pipelining.

If FPGA-based DSP system implementation is to be successfully automated, this requires incorporation of the tools described in Chapter 9, either within commerical HDL-based or C-based design tools (as is beginning to happen with the Synplify DSP tools), or within higher-level design environments such as those outlined in Chapter 11. If this is to be successful, then any high-level approach should allow designers to create efficient FPGA implementations from high-level descriptions, and/or to incorporate existing IP cores, as this may represent considerable years of design time, within the high-level design flow.

The purpose of this chapter is to give some attention to issues which have either had to be ignored or that have not been described in detail. In Section 14.2, reconfigurable FPGA systems fall into this latter category, as the underlying programmable nature of FPGAs provides an interesting platform to allow the realization of such systems. Two topics which need additional comment and which are mentioned in the text in Sections 14.2 and 14.3, are implementation of dedicated floating-point hardware on FPGA and the creation of memory architectures, respectively. In Section 14.4, the chapter also attempts to address future trends and outline future challenge for FPGA developers.

14.2 Reconfigurable Systems

Conventionally, the realm of programmability has been the domain of the microprocessor which executes a sequence of instructions describing the required behaviour of the system; system functionality is then changed by changing the instructions. This has the advantage that it does not require any modification of the circuitry, but is limited in that it cannot deliver the same processing performance and power consumption that comes with a custom hardware solution. The problem is that quite a considerable portion of the circuitry that goes into a modern microprocessor is used

FPGA-based Implementation of Signal Processing Systems R. Woods, J. McAllister, G. Lightbody and Y. Yi
© 2008 John Wiley & Sons, Ltd

for storage and control. This overhead is needed to allow the computational tasks to heavily reuse the small, active portion of the microprocessor, namely the *function units* (FUs).

Conceptually, the FUs are all that are needed to evaluate the operations describing the system. However, in practice, a processor needs to move data between the FUs as well as between the memory and FUs. This is complicated by the practical limitation of how much fast multi-port memory can be placed near the FUs. To address this problem, the memory in a processor is organized into layers of hierarchy. The fastest, most costly memory (the register file) forms the top layer of the memory hierarchy and is placed at the heart of the system to store for values over multiple cycles. With each step down the hierarchy, the memory increases in capacity, but at the cost of slower access speeds. The movement and management of moving data between the layers is another essential task required of the processor.

The result of developing this programmable architecture is that a lot of transistors are not performing any useful computation at one time. In an environment where silicon area is increasingly becoming a premium commodity, there is a strong need to reduce the pressure on large memory blocks as this will lead to bottlenecks. With regard to system memory, there is an effect known as *memory wall*, as proposed by Flynn *et al*. (1989) which indicates that the ratio of the memory access time to the processor cycle time will increase as the technology improves.

This would tend to indicate that a major change is needed in terms of how architectural solutions are derived. To some extent this was seen in the 1980s where damning forecasts on the impact of interconnect delay implied a shift to new architectures, such as those based on systolic arrays architectures (Kung and Leiserson 1979, Kung 1988); one major development was the ill-fated *iWarp* multiprocessing supercomputer developed jointly by Intel and H T Kung's group at Carnegie Mellon University. The aim was to build an entire parallel-computing node in a single microprocessor with the classical organization of systolic arrays, namely with localized memory and nearest-neighbour communications links. Whilst this concept worked well for highly computational operations, such as matrix–matrix and matrix–vector-based computations, it worked less well for less regularly organized computations.

Reconfigurable computing however, does not suffer from this limitation as the main attraction is to change the hardware based on the computational needs. It is targeted at complex systems with the aim to organize functionality in such a way that the computationally complex aspects are decomposed into the field-programmable hardware; reconfigurable computing allows the acceleration seen in the earlier sections of this book to be achieved with the more control-orientated aspects being mapped to a more suitable platform, namely a processor. This infers the use of programmable processors connected to single or arrays of FPGAs.

14.2.1 Relevance of FPGA Programmability

In the early days of FPGAs, a number of different programming technologies emerged specifically E^2PROM technology, antifuse technology and of course, SRAM technology. SRAM programming technology brought about an interesting mode of operation, namely that the functionality could be changed as part of the normal mode of operation or in other words, *reconfigured*. This could either be done *statically* between downtimes in normal modes of operation or *dynamically*, i.e. as part of the normal mode of operation. Basically, the FPGA comprise a huge amount of programmable logic, registers, memory blocks and dedicated processing blocks which can be configured to work in different ways to realize a variety of functions. The FPGA can be considered to be like a smart memory device where the 'state' of the structure is downloaded from a processor to the FPGA device. This configuration of the FPGA is then used to perform an operation on the incoming data. By rewriting different data to the FPGA device, the function performed on the data is changed. So, rather than writing and reading data to a memory device, we are storing data to the FPGA device which changes the function of the data fed into, and accepted back from, the FPGA device.

This is designated as comprising two distinct strata by (Maxfield 2004), namely the *logic stratum* which performs the operation as outlined above and stores the necessary information for the data in the available memories and the *SRAM configuration stratum* which contains the necessary programming information. Thus, interacting with the logic stratum is a normal mode of operation whereas programming the *SRAM configuration stratum* defines the mode of configuration. To use this as a programming mode is highly attractive, meaning it is possible to program the hardware to best meet the computational requirements under consideration, ideally on demand, as and when required.

This concept remains a key attraction as it meant that the available hardware resource could be configured to most efficiently implement the functionality required, but this presents a number of challenges. These include the impact of the time to reconfigure the hardware in environments such as DSP, where data is continually being fed and thus must be stored. In addition, there is the impact in processing time, as indicated by the graph in Figure 14.1 taken from Heron *et al*. (2001) which shows that the reconfiguration time T_R can have an impact on the performance capability, even though the performance rate of the FPGA is superior. Thus there is a break-even time, T_B after which it becomes advantageous to perform reconfiguration. The underlying reliability of a system is not in question as the state of the device is constantly being changed; the question must be asked if the hardware has configured properly.

14.2.2 Existing Reconfigurable Computing

There has been a lot of work on reconfigurable computing platforms, with a good reviews available (Bondalapati and Prasanna 2002, Compton and Hauck 2002, Todman *et al*. 2005). The classification of reconfigurable systems was highlighted initially in (Compton and Hauck 2002) and revised in Todman *et al*. (2005). A simplified version of the diagram shown in Todman *et al*. (2005) is given in Figure 14.2. The figure shows how reconfigurable units, typically in the form of FPGAs, can be added to conventional CPUs to provide the CPU/FPGA configuration.

In the first class shown in Figure 14.2(a), the reconfigurable unit is seen as an external processing unit. A number of example boards exist for this type of configuration from both main FPGA vendors, and also companies such as Celoxica. The second and third classes use a reconfigurable unit as a dedicated co-processor unit, either connected solely to the bus or to both the CPU and the bus. Research examples include RAPid (Ebeling *et al*. 1996) and Piperench (Laufer *et al*. 1999). The fourth class incorporates the reconfigurable unit or fabric within the CPU. This would either be in the form of a fabric to change the interconnectivity of the individual processor

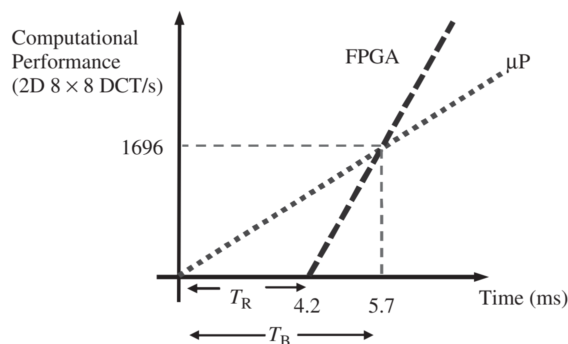


Figure 14.1 Impact of reconfiguration time for an 8 × 8 2D DCT implementation

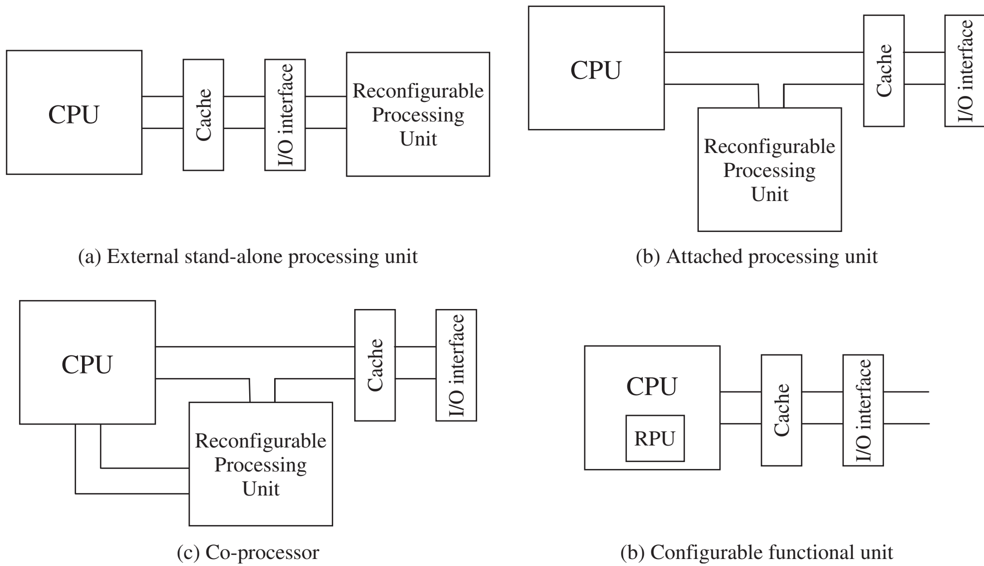


Figure 14.2 Classes of reconfigurable systems

or could involve the reconfiguration of a subunit to change its functionality or individual interconnectivity, or both. An example is the reconfigurable algorithm processing (RAP) technology from Elixent (2005).

This comparison now is a little dated as the evolution in FPGA technology, as described in Chapter 5, indicates. FPGAs are now available with both software and hard microprocessors included on the FPGA substrate. This means that the concept of using FPGAs as a hardware accelerator can still exist, as demonstrated in some commercial and high-performance computing solutions, but now the concept of incorporating the core within the FPGA fabric exists. However, many of the issues addressed by the models still apply as many of the approaches considered the implications in terms of performance with regard to the ratio of communications/computing hardware, both in terms of speed and cost.

14.2.3 Realization of Reconfiguration

There are a number of ways that this reprogrammability can be realized where each approach has subtle, but important differences. The first lies in how the reprogrammability is used in the design to allow a degree of flexibility to be incorporated into the circuit. A large circuit will quickly become very large if many partially redundant components are incorporated at any particular instant of time. MacBeth and Lysaght (2001) highlight the unnecessary overhead of having a reprogrammable core in a reprogrammable device and propose categorizing these circuits as programmable multi-function cores (PMCs). This can also help FPGA realization close the performance gap with ASICs; in ASICs, circuits have to be developed in such a way that they provide versatility, allowing all possible situations to be covered in the one implementation, thereby trading area and speed for flexibility. As the FPGA fabric has to support reconfiguration by its very nature, these resources could be used to change the circuit (its state) so that the best circuit implementation can operate at any given time. The more tailored the circuit becomes, the more likely it will have to be modified to maintain the advantage, but there is a great benefit in terms of both circuit area and data throughput.

Using the fabric of the FPGA as part of the process of changing the circuit's state, can deliver an area decrease of 21% and a speed increase of 14% (MacBeth and Lysaght 2001).

Run-Time Reconfiguration

FPGAs have a limited capacity to perform logic operations, but reconfiguration can be used to get around this. By time-sharing the FPGAs logic, the FPGA can implement circuit operations that are beyond the area capability of the hardware (Lysaght and Stockwood 1996). This has led to the concept of virtual hardware (Brebner 1999) where the physical resource does not limit the amount of circuitry that can be accommodated on a single device.

Run-time Reconfiguration (RTR) can be used to simplify the design of the hardware (Brebner 1999, MacBeth and Lysaght 2001, Walke *et al*. 2000). When a circuit design is constrained by being used across multiple tasks or by being reduced in size through data folding, it will cause complex timing issues and may require hardware sharing. Within the FPGA, a circuit can time-share portions of the hardware. This can be used to simplify the circuit designs. For example, in a hypothetical system proposed by (Walke *et al*. 2000), three finite impulse response (FIR) poly-phase filters are required with different tap lengths; each filter was designed to time-share a limited number of multipliers, but not all at the same time; reconfiguring the circuitry with the choice of FIR filter to be used, removes the time and control complexity required to combine the three filters together. So when the hardware is required to cover a number of possibilities, for reasons of limited hardware resource or multiple standards, a system can reconfigure to select the parts of the circuit that are required. This has distinct advantages as it obviates the need to develop the complex control circuitry needed to switch the various circuit modes, such as that required in ASIC design.

14.2.4 Reconfiguration Models

A number of models have emerged for reconfiguration.

Single Context

The traditional model of the commercially available FPGA is the single context, which only allows the circuitry to be changed by loading a completely new configuration. This type of model is not best suited for RTR because of the associated time to make changes in the firmware. The speed at which the data can be written is limited by the bandwidth of the configuration data transfer from its source to the FPGA. This can be determined by the width (pins) and speed of the interface on the device and in the way it is interfaced to the system. While reconfiguration data is being written onto the device, the current circuitry cannot be used. Other features have been proposed and added to overcome this bandwidth limitation.

Partially Reconfigurable

Examples of commercial partially reconfigurable FPGAs include the Stratix®III family from Altera and the Virtex™-5 FPGA family from Xilinx. These devices have much more scope for RTR than single context FPGA as sections can be changed independently. As the reconfiguration sections are made smaller, the cost incurred in making changes to the circuitry reduces, thus allowing frequent changes without significant overheads. A feature that is often available with this type of device is reconfiguration with continued operation. While parts of the FPGA are reconfigured, other parts can still be functional, masking the time needed for reconfiguration.

Multi-Context

There have been a number of studies into multi-context designs (DeHon 1996, Scalera and Vazquez 1998, Trimberger *et al*. 1997), but so far this has not resulted in a commercially available product. These devices tackle the problem of transferring the reconfiguration data by storing more than one plane of reconfiguration data on-chip. The process of reconfiguration is achieved by selecting one of the planes to drive the configuration of the logic. The switch between contexts can be achieved in only a few clock cycles. The multiple contexts allow background loading of the configuration data, as the configuration data can be loaded into a context that is not active.

A possible problem with such a device lies in the number of contexts required in a typical system implementation, this may be greater than the available hardware and the sharing of data between contexts. The SCORE (Section 2.6.2) architecture shares some similarities with the multiplier contexts FPGA, but allows the amount of resources used for the contexts to be altered.

Pipeline Reconfiguration

The pipeline reconfiguration model views the reconfiguration of the device as pipelined blocks and can be seen as a modification of partially reconfigurable FPGAs (Luk *et al*. 1997). It is viewed as pipeline datapaths, each stage of which is reconfigured as a whole. This permits the overlapping of the reconfiguration and execution. The required reconfiguration is broken into stages of reconfiguration where each stage is loaded in sequence. After each stage has been programmed, it immediately begins to operate, thereby the configuration of a stage is exactly one step ahead of the dataflow. Once the device has run out of space, it starts to swap out stages that have been resident longest on the FPGA, replacing them in the next stage. This allows applications to exceed the physical resources of the device and still run with a reduced throughput. Piperench (Laufer *et al*. 1999) is an example of this type of model. In this case, the author points out the advantage of forward compatibility, as further devices can retain the same stages and just increase the numbers of stages.

Configuration Caching

Figure 14.3 shows an abstracted model of the FPGA, where there is a FPGA array and a reconfiguration data storage device. As it is not possible to supply the bandwidths required, the movement of configuration data can result in a long halt in the circuit processing. In this case, pre-fetching and caching can be used (Hauck *et al*. 1999), as shown in Figure 14.3, to reduce the 'burstiness' of

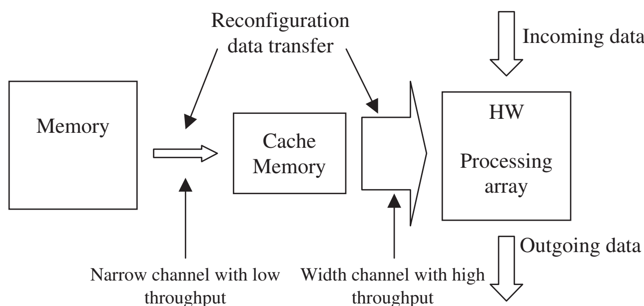


Figure 14.3 Pre-fetching and caching used to get around the bandwidth limitation of reconfiguration data transfer

the data movement. Caching data close to the FPGA's array, such as on-chip, makes it possible to use a high-bandwidth channel with the cache being fed via a narrow channel. However, a problem with pre-fetching occurs with conditional branching when incorrect data is fetched (Hauck *et al*. 1999).

Configuration Compression

Configuration compression reduces the amount of data to be moved and therefore cuts the reconfiguration time by using data compression. Figure 14.4 shows an abstract view of this model. As with the pre-fetching, the link from the decompression hardware to the FPGA array is much larger than the link to the reconfiguration data store. The concept was initially developed for the Xilinx XC6200 FPGA technology which has a wildcarding feature that allows address and data values to be written simultaneously to multiple locations. This device had been developed specifically to support dynamic reconfiguration and it recognized that it is necessary to change parts of the device at the same time, hence the wildcarding features. Hauck *et al*. (1999) showed that it is possible to compress the data streams in a way that allows the wildcarding hardware of the XC6200 to decompress the reconfiguration data stream. Hauck *et al*. (1999) and Li and Hauck (2001) went on to consider other compression methods with low overheads that can be used with other FPGAs.

14.3 Memory Architectures

The support for parallel and pipelining operation was highlighted as the major attraction of FPGAs when considered for implementing DSP systems. However, one factor that has received some attention throughout is the availabilitiy of a wide range of different sizes of parallel memory, whether this was in the form of distributed RAM blocks, simple LUTs or a single flip-flop. As was highlighted by Flynn *et al*. (1989), the *memory wall* gives a depressing future for fixed computer architectures as the ratio of the memory access time to the processor cycle time increases. Whilst some approaches act to address this via technology, FPGAs get around this problem by naturally developing a highly parallel solution through the use of a distributed memory architecture. This can happen through deliberate derivation of a distributed memory architecture or as a result of a algorithmic optimization, as for example in the application of pipelining which is in effect results

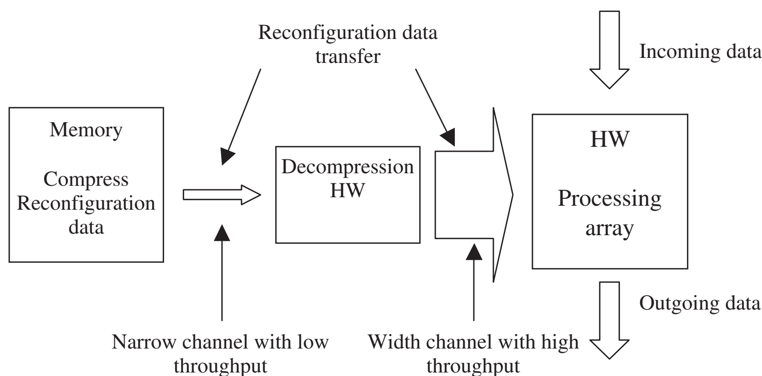


Figure 14.4 Compression used to get around the bandwidth limitation of reconfiguration data transfer

in the creation of distributed memory. This approach is particularly suited for DSP due to data-independent operation and resgular high computation rates, thereby allowing parallel architectures with little control complexity.

This would then suggest that FPGAs provide a useful platform for developing new types of computer architectures which are memory-orientated, rather than computation-orientated. The text has clearly shown how different circuit architectures were developed for different DSP algorithms. The architectrual derivations were usually driven by creating the necessary computation resources to meet the detailed specifications of the system. There is a strong case for suggesting more detailed research into the development of memory-orientated architectures where some basic principles are developed for creating the memory requirements of some DSP algorithms. This was seen to some extent in the Imagine processor (Kapasi *et al*. 2002) where the memory was developed for the class of algroithms needed, and in trivial FPGA examples in Chapters 6, 8 and 9, where different memory, i.e. LUTs in the forms of SRLs, was selected in preference to flip-flops to provide more efficient implementation delay chains, either because of lack of flip-flop resource or more relevant selection of resource. However, this has tended to be a good design decision rather than a conscious need to develop memory-orientated architectures. Work by Fischaber *et al*. (2007) has suggested how design of memory can be directed from the dataflow level.

14.4 Support for Floating-point Arithmetic

A conscious decision to first introduce scalable adder structures in early FPGAs and then dedicated multiplicative complexity in latter versions, such as the such as the Stratix®III family from Altera and the Virtex™-5 FPGA family from Xilinx, has really influenced the use of FPGAs for DSP systems. Along with the availability of distributed memory, this has driven an additional interest in using FPGAs for supercomputing due to the extremely high computation rates required. A number of hardware platforms are now available from a wide range of vendors including solutions from Cray, SGI and Nallatech, and would appear to offer high performance for supercomputing. However, work by Craven and Athanas (2007) would suggest that the performance achievable, even when using many of the techniques highlighted in this book, is limited and that use of FPGA in supercomputing applications, will have limited application.

A key reason for this is the lack of suitable floating-point support in FPGA, even though the work in Craven and Athanas (2007) avoided the use of floating-point arithmetic and used fixed-point hardware. The figures outlined in Chapter 3 and repeated here in Table 14.1, highlight the problems of using floating-point arithmetic in FPGAs. A fixed-point implementation would only need one DSP48 if the wordlength was less than 18 bits, and would not require a lot of the flip-flops outlined in the table. This extra hardware is required to implement the hardware necessary to perform the data selection, rounding and normalization, illustrated earlier in Figure 3.10. If FPGAs are to be

Table 14.1 Area and speed figures for various floating-point operators implemented using Xilinx Virtex-4 FPGA technology

Function	DSP48	LUT	Flip-flops	Speed (MHz)
Multiplier	4	799	347	141.4
Adder		620	343	208.2
Reciprocal	4	745	266	116.5

Table 14.2 Impact of technology scaling on interconnect delay (Davis *et al*. 2001)

Technology	MOSFET switching delay (ps)	Intrinsic delay of 1mm interconnect (ps)	
		Minimum scaled	Reverse scaled
1.0 μm(Al, SiO_2)	20	5	5
0.1 μm(Al, SiO_2)	5	30	5
35 nm(Cu, low k)	2.5	250	5

used extensively in supercomputing, support for these functions will need to be included in future versions of FPGAs as well as much better high-level programming support for pipelining.

14.5 Future Challenges for FPGAs

The offerings in the most recent commercial FPGAs are highly impressive. The recent devices represent highly complex platforms which push the silicon technology and now apparently are used, in preference to memory, as complex devices to test future silicon technologies. However, there are number of challenges that will particularly affect FPGAs.

Technology scaling offers a number of advantages, but it accentuates one particular relationship which is problematic for FPGAs and that is the ratio of interconnect to transistor delay. Some examples of this are shown in Table 14.2 which is taken from (Davis *et al*. 2001) and shows the ratio of interconnect to transistor delay for various technologies. In this table, *minimum scaling* represent what will actually happen to the delay if no measures are taken into consideration whereas *reverse scaling* refers to the impact of *fattening* the wires, thereby reversing the *scaling* process. Reverse scaling will be deliberately employed to counteract the increased resistance impact of the interconnect, as indicated in (Table 14.2). Whilst this addresses the timing problems, it causes a dramatic decrease in wiring densities, thereby reducing levels of integration and making larger bus-driven structures inefficient from an area perspective. Given the high dependence of FPGAs on programmable routing, either approach has implications; one, *minimum scaling*, will act to slow the system considerably whereas the other, *reverse scaling*, will have a major area impact. It seems likely that the impact of this effect will be to increase the use of larger heterogeneous blocks on FPGAs.

Progressive scaling has driven the semiconductor industry, allowing vendors to deliver faster, cheaper circuits with ever-increasing functionality. A more serious issue than increasing impact due to interconnect delay, is the impact of process variation. With technology at 40 nm and the shift towards sub-10 nm by 2018 (Semiconductor Industry Association 2005), variability in device characteristics now represents a major challenge in delivering next-generation SoC systems, including FPGAs. With other problems it has been possible to adopt the worst case and then use other approaches to overcome the limitations, but with process variation, questions arise about the reasons for investing in next-generation scaling.

References

Bondalapati K and Prasanna V (2002) Reconfigurable computing systems. *Proc. IEEE* **90**(7), 1201–1217.

Brebner G (1999) Tooling up for reconfigurable system design *Proc. IEE Coll. on Reconfigurable Systems*, pp. 2/1–2/4.

Compton K and Hauck S (2002) Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys* **34**(2), 171–210.

Craven S and Athanas P (2007) Examining the viability of fpga supercomputing. *EURASIP Journal on Embedded Systems* **1**, 13–13.

Davis J, Venkatesan R, Kaloyeros A, Beylansky M, Souri S, Banerjee K, Saraswat K, Rahman A, Reif R and Meindl J (2001) Interconnect limits on gigascale integration (gsi) in the 21st century. *Proc. IEEE* **89**, 305–324.

DeHon A (1996) Dynamically programmable gate arrays: A step toward increased computational density *Proc. 4th Canadian Workshop on Field-Programmable Devices*, pp. 47–54.

Ebeling C, Cronquist DC and Franklin P (1996) Rapid: Reconfigurable pipeline datapath *Proc. 6th Int. Workshop on Field-Programmable Logic and Compilers*, pp. 126–135.

Elixent (2005) Reconfigurable algorithm processing (rap) technology. Web publication downloadable from http://www.elixent.com/.

Fischaber S, Woods R and McAllister J (2007) Soc memory hierarchy derivation from dataflow graphs *IEEE Workshop on Signal Processing Systems*, pp. 469–474.

Flynn MJ, Hung P and Rudd K (1989) Deep-submicron microprocessor design issues. *IEEE Micro* **19**, 11–22.

Hauck S, Li Z and Schwabe EJ (1999) Configuration compression for the xilinx xc6200 fpga. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems* **18**(8), 1107–1113.

Heron J, Woods R, Sezer S and Turner RH (2001) Development of a run-time reconfiguration system with low reconfiguration overhead. *Journal of VLSI Signal Processing, Special issue on Re-configurable Computing* **28**(1/2), 97–113.

Kapasi U, Dally W, Rixner S, J.D. O and Khailany B (2002) Virtex5.pdf *Proc. 2002 IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors*, pp. 282–288.

Kung HT and Leiserson CE (1979) Systolic arrays (for vlsi) *Sparse Matrix Proc. 1978*, pp. 256–282.

Kung SY (1988) *VLSI Array Processors*. Prentice Hall, Englewood Cliffs, NJ.

Laufer R, Taylor R and Schmit H (1999) PCI-Piperench and the SWORDAPI – a system for stream-based reconfigurable computing. *Proc. IEEE Symp. on Field-programmable Custom Computing Machines*, Napa, USA, pp. 200–208.

Li Z and Hauck S (2001) Configuration compression for virtex fpgas *Proc. IEEE Conf. on FPGA-based Custom Computing Machines*, pp. 147–159.

Luk W, Shirazi N, Guo SR and Cheung PYK (1997) Pipeline morphing and virtual pipelines. *7th Int. Workshop on Field-programmable Logic and Applications*, pp. 111–120.

Lysaght P and Stockwood J (1996) A simulation tool for dynamically reconfigurable field programmable gate arrays. *IEEE Trans. on VLSI Systems* **42**(2), 381–390.

MacBeth J and Lysaght P (2001) Dynamically reconfigurable cores *Proc. IEEE Conf. on FPGA-based Custom Computing Machines*, pp. 462–472.

Maxfield C (2004) *The Design Warrior's Guide to FPGAs*. Newnes, Burlington.

Scalera SM and Vazquez JR (1998) The design and implementation of a context switching fpga *Proc. IEEE Conf. on FPGA-based Custom Computing Machines*, pp. 78–85.

Semiconductor Industry Association (2005) International technology roadmap for semiconductors: Design. Web publication downloadable from http://www.itrs.net/Links/2005ITRS/Design2005.pdf.

Todman T, Constantinides G, Wilton S, Cheung P, Luk W and Mencer O (2005) Reconfigurable computing: architectures and design methods. **152**(2), 193–205.

Trimberger S, Carberry D, A. J and Wong J (1997) A time-multiplexed fpga *Proc. IEEE Conf. on FPGA-based Custom Computing Machines*, pp. 22–28.

Walke R, Dudley J and Sadler D (2000) An fpga based digital radar receiver for soft radar *Proc. 34th Asilomar Conf. on Signals, Systems, and Computers*, pp. 73–78.

Index

2nd order IIR filter, 178, 195

Actel® Antifuse SX FPGA, 106
Actel® FPGA, 105
adaptive algorithms, 277
adaptive beamformer, 271
adaptive beamforming, 271
adaptive delayed LMS filter, 199
adaptive filtering, 27, 277
adaptive filters, 28
adders, 43
adaptive differential pulse code modulation (ADPCM), 29
as late as possible (ALAP), 172
allocation, 172
adaptive logic module (ALM), 85
Altera FPGAs, 82
Amphion FFT core, 347
architecture, 58
architecture design, 287
architecture development, 282
arithmetic libraries, 301
arithmetic requirements, 61
advanced RISC machine (ARM) microprocessor, 63
as soon as possible (ASAP), 172
application specific integrated circuit (ASIC), 74
Atmel® AT40K FPGA, 108, 109
Atmel® FPGAs, 108

balance equations, 246
beamformer, 266, 271
beamformer design, 323
behavioural synthesis, 171
bilinear transform, 24
binding, 172
boundary cell, 303

carry-save adder, 46
carry-save multiplier, 46
cell latency, 306
cell timing, 305, 306
circuit architecture, 58
complex instruction set computer (CISC), 62
configurable logic block (CLB), 95, 96
Clearspeed CSX600 processor, 72
clock networks, 89, 102
clock rate, 144
clock tree, 334
clustering, 251
Connection Machine 2 (CM-2), 69
code generation, 253
complex multiplication, 303
computational complexity, 59
control of QR architecture, 314
control wrapper, 260
COordinate Rotation Digital Computer (CORDIC), 38, 342
correlation, 23
critical loop, 186
cyclo-static dataflow (CSDF), 246
cut-set theorem, 154
cyclo-static dataflow, 246

data independency, 61
data re-ordering, 337
datapath delay pair, 187
delay scaling, 154, 155
delay transfer, 154

FPGA-based Implementation of Signal Processing Systems R. Woods, J. McAllister, G. Lightbody and Y. Yi
© 2008 John Wiley & Sons, Ltd

dependence graph, 283
design reuse, 125, 180
DFG actor, 254
dataflow graph (DFG), 149
discrete cosine transform (DCT), 18, 123
discrete Fourier transform (DFT), 17
discrete wavelet transform, 19
distributed arithmetic (DA), 117, 340
distributed RAMs, 115
division by functional iteration, 47
dataflow process network (DPN), 245
digital signal processing (DSP) function blocks, 87
DSP processing blocks, 87
DSP system mapping, 149
DSP48E, 100
dynamic power consumption, 331

Electroencephalography (EEG), 14
extended MARS (EMARS) algorithm, 193
energy, 331
exponent, 42
external memory interfaces, 91, 103

fast Fourier Transform (FFT), 16, 347
finite impulse response (FIR) filter, 20, 151, 344
fixed beamformer, 266
fixed coefficient, 116, 338
fixed-point arithmetic, 41
Flynn's taxonomy, 62
folding, 165, 195
force directed scheduling, 173
field programmable gate array (FPGA), 1
FPGA challenges, 9
FPGA definition, 80
FPGA evolution, 6
FPGA I/O, 103
FPGA memory organisation, 99
functional engine, 260
Fusion™, 105

generic architecture, 292, 301
generic control, 319
generic QR cells, 319
Gibbs phenomenum, 23
Gigabit transceivers, 91
Givens rotations, 280, 285
graph balancing, 250
graph level optimizations, 250

Hardcopy® Structured ASIC, 92
hardware sharing, 163, 190
Harvard architecture, 64
heterogeneous system prototyping, 247
hierarchical retiming, 188
hierarchy, 179
homogeneity, 22

ILLIAC IV, 69
Imagine processor, 70
impulse invariant method, 24
infinite impulse Response (IIR) filter, 23, 157
integrated circuit (IC), 4
interleaving, 147, 258
internal cell, 303
intellectual property (IP) cores, 213
IP core design process, 272
iteration bound, 191, 193
iterative refinement scheduling, 173

Jewett waveform, 20

latency, 15, 146
lateral delay scaling, 262
lattice filter, 160
Lattice® ispXPLD 5000MX FPGA, 103
least mean squares (LMS), 30
least significant bit (Isb), 38
linear architecture, 310
linear array, 296, 306
linear phase, 23
linear programming, 188
linear time invariant (LT1), 22
LMS algorithm, 31
LMS filter, 199
LMS versus RLS, 278
locality, 343
logarithmic number systems (LNS), 38
logic delay measurement circuit (LMDC), 337
longest path matrix algorithm, 158
loop search, 191

Makimoto's wave, 8
mantissa, 42
Minnesota architecture synthesis (MARS) algorithm, 191
match z-transform, 24
matrix multiplication, 257
MAX® FPGA technology, 83

MAX®II FPGA technology, 83
microchip, 4
microcontrollers, 57
microprocessors, 62
modular design procedure, 174
Moore's Law, 2
most significant bit (msb), 38
motion estimation, 116
multi-rate DFGs (MRDFG), 255
multidimensional array DF (MADF), 259, 263
multidimensional SDF, 246
multiply-accumulate, 178
multiple-input multiple-data (MIMD), 75

noise cancellation, 30
non-recurrent engineering (NRE) costs, 5
non-restoring recurrence algorithm, 48
normalized lattice filter, 264
Noyce, 4

one's complement, 38
orthogonal frequency division multiplexing (OFDM), 17
overflow, 41

parallel FIR filter, 161
parallelism, 60, 146
parameterizable QR architecture, 307
phase lock loop (PLLs), 91, 104
pipeline cores, 254
pipelining, 3, 15, 66, 147, 276, 339
pipelining period, 158, 176, 182, 185, 193
power reduction, 335
ProASIC^{PLUS}, 105
processor classification, 61
processor latency, 175
programmability, 3, 8, 58
programmable array logic, 81
programmable logic device (PLD), 81
programmable shift register, 114

QR architecture, 307
QR decomposition, 278
QR factorization, 280

read only memory (ROM), 81
rectangular architecture, 316
rectangular array, 298
recurrence division, 47
recursive least squares (RLS), 30
Reduced coefficient multiplier, 120
reduced instruction set computer (RISC), 65
redundant representation, 39
relationship inputs vector, 187
repetitions vector, 246
residue number systems (RNS), 38
retiming, 150, 176, 185, 301, 306
RLS algorithm, 32
RLS filtering, 278

sampling rate, 14, 144
saturation, 41
scalable, 275
scaling, 41
scheduling, 252, 290
scheduling algorithms, 172
second order IIR filter, 193
security, 92
shift invariance, 22
shift register, 113
sign, 42
signal flow graph (SFG), 150, 283
signal to noise ratios, 55
signed digit number representations, 38, 39
signed magnitude, 38
single-input multiple-data (SIMD), 69
sources of power consumption, 330
space-time data format, 174
sparse linear architecture, 310
sparse linear array, 297
sparse rectangular architecture, 316
sparse rectangular array, 300
square root, 48
squared Givens rotation, 287
static power consumption, 330, 332
Storm processor, 71
Stratix, 342
Stratix® III FPGA, 85
sub matrix decomposition, 259
super Harvard architecture (SHARC), 68
superposition, 21
switched capacitance, 337
synchronous dataflow (SDF), 245
systolic arrays, 67, 285

throughput rate, 144
TILE64 processor, 74
TMS32010, 65
TMS320C64xx, 67

treble weighted graph, 188
triangular array, 293
triple data encryption standard (DES), 342
two's complement, 39

unfolding, 163

vertical delay scaling, 262
very long instruction set (VLIW), 68
Virtex™ FPGA, 95
Virtex™-5 FPGA, 97, 100
virtual processor, 260
voltage scaling, 335
von Neumann, 62, 64

walking one, 112
Wallace tree multiplier, 47, 176
wave digital filter, 183, 188
wavelet transform, 19
white box, 180, 260
Wiener filter, 28
windowing, 23
wordlength, 276

XC2064, 5
Xilinx FFT core, 347
Xilinx FPGA technology, 93
Xilinx Virtex™-5, 94
Xilinx XC2000, 79
Xpower, 340