

FPGA-BASED PROTOTYPING METHODOLOGY MANUAL

Best practices in Design-for-Prototyping

Doug Amos
Austin Lesea
René Richter

FPGA-BASED PROTOTYPING METHODOLOGY MANUAL

Doug Amos, Austin Lesea and René Richter



"ARM has developed its own FPGA boards for prototyping for over a decade and uses these alongside simulation and emulation for validation of early operating system ports and for improving quality and confidence. The FPMM is recommended reading to anybody considering FPGA prototyping as part of their validation methodology, especially for proving their IP and for early software development."

Spencer Saunders, Engineering Manager — Platforms
ARM



"The modular nature of the Synopsys HAPS systems coupled with our Design-for-Prototyping approach has allowed LSI Corporation to do pre-silicon development on multiple SoC projects so far. If you are looking to benefit from FPGA-Based Prototyping then the FPMM is a great place to start, for management and engineers alike. Even experienced prototypers will find the FPMM a source of inspiration and guidance. I wish we'd had this when we started!"

Brian Nowak, Senior Integration Engineer
LSI Corporation



nVIDIA

"At NVIDIA we have benefited from FPGA-Based Prototyping for over a decade. In all cases we have been able to validate functionality and exercise software months in advance of the first silicon being available. The FPMM book and accompanying online community will educate our industry in the methodology of prototyping and share the latest techniques in implementation and leading edge technologies. I recommend the FPMM to anybody considering prototyping as a validation vehicle for developing silicon products."

Fernando Martinez, Engineer, Mobile R&D
NVIDIA Corporation

**SYNOPSYS®**
Press

eBook Edition
Not for resale

FPGA-Based Prototyping Methodology Manual

Best Practices in Design-for-Prototyping

Doug Amos
Synopsys, Inc.

Austin Lesea
Xilinx, Inc.

René Richter
Synopsys, Inc.

SYNOPSYS®

XILINX®

Doug Amos
Synopsys, Inc.
Reading
United Kingdom

Austin Lesea
Xilinx, Inc.
San Jose, CA
USA

René Richter
Synopsys, Inc.
Erfurt
Germany

Library of Congress Control Number: 2011920198
Hardcover ISBN 978-1-61730-003-5
Paperback ISBN 978-1-61730-004-2
eBook ISBN 978-1-61730-005-9

Copyright © 2011 Synopsys, Inc. All rights reserved.
Portions Copyright© 2009-2011 Xilinx, Inc. Used by permission.
Portions Copyright© 2011 ARM Limited. Used by permission.

This work may not be translated or copied in whole or in part without the written permission of Synopsys, Inc. (700 E. Middlefield Road, Mountain View, CA 94117 USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden. The use in this publication of trade names, trademarks, service marks and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Published by Synopsys, Inc., Mountain View, CA, USA
http://www.synopsys.com/synopsys_press

Publishing services provided by Happy About®
<http://www.happyabout.com>

Printed in the United States of America
February 2011

TRADEMARKS

Synopsys, the Synopsys logo, ARC, Certify, Design Compiler, DesignWare, HAPS, HapsTrak, Identify, Synplicity, Synplify, Synplify Pro, and VCS are registered trademarks or trademarks of Synopsys, Inc.

LSI is a trademark of LSI Corporation. The LSI logo is a registered trademark of LSI Corporation.

NVIDIA and the NVIDIA logo are registered trademarks of NVIDIA Corporation in the United States and other countries.

Xilinx, Inc., XILINX, the Xilinx logo, ChipScope, CORE Generator, ISE, MicroBlaze, Spartan, System ACE, and Virtex, are trademarks of Xilinx in the United States and other countries.

ARM, the ARM logo, and AMBA are registered trademarks of ARM Limited. AHB, APB, AXI, Cortex, and Mali are trademarks of ARM Limited. “ARM” is used to represent ARM Holdings plc; its operating company ARM Limited; and its regional subsidiaries. ARM, the ARM logo, and AMBA are registered trademarks of ARM Limited.

All other brands or product names are the property of their respective holders.

DISCLAIMER

All content included in this FPGA-Based Prototyping Methodology Manual is the result of the combined efforts of Synopsys, Inc., Xilinx, Inc., and other named contributors. Because of the possibility of human or mechanical error, neither the authors, contributors, Synopsys, Inc., Xilinx Inc, nor any of their affiliates guarantees the accuracy, adequacy or completeness of any information contained herein and is not responsible for any errors or omissions, or for the results obtained from the use of such information. THERE ARE NO EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE relating to the FPGA-Based Prototyping Methodology Manual. In no event shall the authors, contributors, Synopsys, Inc., Xilinx, Inc., or their affiliates be liable for any indirect, special or consequential damages in connection with the information provided herein.

THE XILINX HARDWARE FPGA AND CPLD DEVICES REFERRED TO HEREIN (“PRODUCTS”) ARE SUBJECT TO THE TERMS AND CONDITIONS OF THE XILINX LIMITED WARRANTY WHICH CAN BE VIEWED AT <http://www.xilinx.com/warranty.htm>. THIS LIMITED WARRANTY DOES NOT EXTEND TO ANY USE OF PRODUCTS IN AN APPLICATION OR ENVIRONMENT THAT IS NOT WITHIN THE SPECIFICATIONS STATED IN THE XILINX DATA SHEET. ALL SPECIFICATIONS ARE SUBJECT TO CHANGE WITHOUT NOTICE. PRODUCTS ARE NOT DESIGNED OR INTENDED TO BE FAIL-SAFE OR FOR USE IN ANY APPLICATION REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS LIFE-SUPPORT OR SAFETY DEVICES OR SYSTEMS, OR ANY OTHER APPLICATION THAT INVOKES THE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR PROPERTY OR ENVIRONMENTAL DAMAGE (“CRITICAL APPLICATIONS”). USE OF PRODUCTS IN CRITICAL APPLICATIONS IS AT THE SOLE RISK OF CUSTOMER, SUBJECT TO APPLICABLE LAWS AND REGULATIONS.

Foreword

We are grateful to Helena Krupnova of STMicroelectronics for sharing her insight and experience of FPGA-based prototyping in this foreword.

Grenoble – December 2010

In this book are introduced the “Three Laws of Prototyping.”

For the last ten years here at STMicroelectronics in Grenoble, France, I’m working within a hardware-based verification team providing FPGA-based prototyping solutions for the use of software development teams worldwide and we have learnt that there are many steps and obstacles for the unwary prototyper. This book covers most of them, but they all distil down to these three “laws”:

- SoCs are larger than FPGAs
- SoCs are faster than FPGAs
- SoC designs are FPGA-hostile

And yet, FPGAs provide a platform for SoC development and verification unlike any other and their greatest value is in their unique ability to provide a fast and accurate model of the SoC in order to allow pre-silicon validation of the embedded software.

Like all leading-edge developers, we at STMicroelectronics use a wide range of different tools and platforms during SoC design and verification. There is a prototyping community within ST, including several teams belonging to different divisions and addressing different design environments with a variety of prototyping approaches – real-time prototyping, commercial solutions, and custom machines. We use FPGA-based prototypes because they are often more than ten-times faster than emulators and this allows us to respond to the needs for a fast platform for interactive testing of new SoC software by a large number of developers on different sites.

Embedded software has become the dominant part of the effort in modern SoC design. We need to start earlier in our project just to allow enough time for the completion of the software and this means running it on the SoC “pre-silicon.” It is very clear to me that the main benefit from FPGA-based prototyping is precisely for the software teams. Prototyping is the hardware team’s answer to the software team’s problems.

It is simply the only type of technology that offers realism to pre-silicon software developers by, for example, giving instantaneous response from JTAG debuggers, allowing operating system booting in near real-time and allowing real-time

interfaces to the system peripherals. I could go on but you can read for yourselves the benefits that others have obtained from using FPGA-based prototyping in the introductory chapters of this book.

This book discusses different possible board and system architectures and I believe that learning from previously available architectures is mandatory for everybody building high-speed prototypes. The make versus buy decision criteria, both technical and commercial, are also to be found in this book reinforcing in my mind my belief that the only reason to build a custom platform in-house is when a high number of replicated platforms need to be built. How many is a “high number?” There is help in these pages for the reader to work this out for themselves, but in general only dozens of copies of the board will allow users to get the return on the engineering cost invested for the board’s development. Certainly, for any single usage, buying a commercial solution is much more economical!

The obstacles to FPGA-based prototyping that I mentioned in my opening make it daunting for some. However, there are also a number of steps in overcoming those three laws of prototyping that can be taken by us all. These are outlined in this comprehensive book and if you read this before embarking on your project, especially the Design-for-Prototyping guidelines, then you are more likely to succeed.

The book ends with a discussion of linking prototypes to simulation tools via transaction-level interfaces and leads into a glimpse of future directions. I believe that we will see more integration of FPGA-based prototyping with ESL modeling to create transaction-based platforms which will extend the benefit to software developers from before RTL right through FPGA-based prototyping to first silicon.

This book, and its companion web-based forum, will attract managers’ attention to the importance of prototyping and I expect it will be a great support to the prototyping community!

Helena Krupnova

Prototyping Team Leader
ST Microelectronics
Grenoble, France

The aim of this book

This book is a seed.

It is a seed around which the sum knowledge of FPGA-based prototyping can crystallize.

We have gathered input from far and wide in order to present a snapshot of best practices in the field of FPGA-based prototyping. We hope that there is something in this manual for everyone but undoubtedly, owing to time and space constraints, there is probably something missing for everyone too.

The book in your hand (or the eBook on your screen) represents a first edition of the FPMM. From an early stage, we have been planning that this book will have a parallel on-line component, in which we can add or correct information, provide download pdf copies and build upon each reader's experience to build further editions as required. In this way, the FPMM will thrive on feedback and in turn can provide even more in-depth education for prototypers around the world. Using new media we can help to unite otherwise isolated and outnumbered prototyping experts into a respected forum of real value. This value will not just benefit the prototypers but also the SoC teams of which they will be an increasingly important part, promoting FPGA-based prototyping into its rightful place as a serious verification and validation methodology.

We hope you like the book and we look forward to seeing you on the FPMM on-line community soon (go to www.synopsys.com/fpmm).

Austin, Doug and René

January, 2010

A note from the publisher

Thank you for your interest in this technical series book from Synopsys and our partners. We at Synopsys Press are excited to introduce the FPGA-Based Prototyping Methodology Manual (FPMM), which, like our previous successful Methodology Manuals (e.g., *Verification Methodology Manual for Low Power* (VMM-LP), aims not only to educate practitioners but also inform leaders.

We are particularly pleased that so many experts were involved in the creation of the contents and for the extensive peer review. At the end of each chapter we acknowledge to those who made a significant contribution to its content.

To learn more about this Synopsys Press book and the others in both the technical and business series, please visit www.synopsys.com/synopsys_press.

We hope you enjoy the book,

Phil Dworsky

Publisher,
Synopsys Press,
February, 2011

The book's organization

The book is organized into chapters which are roughly in the same order as the tasks and decisions which are performed during an FPGA-based prototyping project.

Readers will be approaching this book from a number of directions. Some will be experienced with many of the tasks involved in FPGA-based prototyping but are looking for new insights and ideas; others will be relatively new to the subject but experienced in other verification methodologies; still others may be project leaders who need to understand if the benefits of FPGA-based prototyping apply to their next SoC project. So, depending on your starting point you may need to start reading the book in a different place. In anticipation of this, we have tried to make each subject chapter relatively standalone, or where necessary, make numerous forward and backward references between subjects, and provide recaps of certain key subjects

Chapters 1&2: We start by analyzing the complexity of the problem of validating an SoC and the software embedded within it. We introduce a number of different prototyping methods, not just FPGA. We then go on to describe the benefits of FPGA-based prototyping in general terms and give some real-life examples of successful projects from some leading prototypers in the industry.

Chapter 3: This is a primer on FPGA technology and the tools involved, giving a new perspective on both in the context of FPGA-based prototyping. Experienced FPGA users may feel that they can skip this chapter but it is still recommended as a way to look at FPGAs from possibly a new viewpoint.

Chapter 4: Every journey begins with a single step. After hopefully whetting the reader's appetite for FPGA-based prototyping, this chapter brings together sufficient information to get us started, allowing us to gauge the effort, tools and time needed to create a prototype.

Chapters 5 and 6: The hardware component of a prototype should be chosen early in the project. These chapters give guidance on how to best create a platform in house, or how to choose between the many commercial platforms and how to make an informed comparison between them (see also Appendix B)

Chapter 7, 8, 9 and 10: Key information on manipulating a design to make it ready for implementation in FPGA hardware, with special focus on RTL changes, partitioning and IP handling. There is also guidance on how a Design-for-Prototype SoC design style can be adopted to make designs more suitable for FPGA-based prototyping team.

Chapters 11&12: The board is ready; the design is ready; what happens when the two are put together? These chapters cover how to bring up the prototype in the lab and then go on to debug the RTL and software on the system and make fast

iterations of the design. There is also a discussion of the deployment of the prototype outside the lab.

Chapter 13: We have a working FPGA-based prototype; what else can be done with such a useful platform? This chapter shows the benefit of tailoring the prototype to be used within wider verification environments including RTL simulators and SystemC™-based virtual models.

Chapter 14: Here we perform some future-gazing on FPGA-based prototyping and beyond into a hybrid prototyping, taking concepts from chapter 13 and other places to some new conclusions.

Chapter 15: This then leads into some conclusions and re-iteration of key rules and suggestions made throughout the manual.

Appendix A: Here is an instructive worked example of a recent FPGA-based prototype project performed at Texas Instruments giving details of the various steps taken and challenges overcome.

Appendix B: There is also an economic and business comparison between making prototype hardware in-house ‘v’ obtaining them commercially.

NOTE: the majority of the FPMM contents are intended to be generic and universally applicable, however, where examples are given, we hope that readers will forgive us for using tool and platforms best known to us at Synopsys® and Xilinx® (i.e., our own).

Table of Contents

Foreword	v
CHAPTER 1 Introduction: the challenge of system verification.....	1
1.1. Moore was right!.....	1
1.1.1. SoC: A definition . . . for this book at least.....	1
1.2. The economics of SoC design	2
1.2.1. Case study: a typical SoC development project	4
1.3. Virtual platforms: prototyping without hardware.....	6
1.3.1. SDK: a very common prototyping environment	7
1.3.2. FPGA: prototyping in silicon . . . but pre-silicon.....	8
1.3.3. Emulators: prototyping or verification?.....	9
1.3.4. First silicon as a prototype platform	10
1.4. Prototyping use models	10
1.4.1. Prototyping for architecture exploration.....	11
1.4.2. Prototyping for software development	11
1.4.3. Prototyping for verification.....	12
1.5. User priorities in prototyping.....	13
1.6. Chip design trends.....	15
1.6.1. Miniaturization towards smaller technology nodes.....	15
1.6.2. Decrease in overall design starts	16
1.6.3. Increased programmability and software.....	17
1.6.4. Intellectual property block reuse	19
1.6.5. Application specificity and mixed-signal design	21
1.6.6. Multicore architectures and low power	22
1.7. Summary.....	23
CHAPTER 2 What can FPGA-based prototyping do for us?	25
2.1. FPGA-based prototyping for different aims.....	25
2.1.1. High performance and accuracy	26
2.1.2. Real-time dataflow	27
2.1.3. Hardware-software integration.....	28
2.1.4. Modeling an SoC for software development	28

2.1.5. Example prototype usage for software validation	30
2.2. Interfacing benefit: test real-world data effects	33
2.2.1. Example: prototype immersion in real-world data.....	34
2.3. Benefits for feasibility lab experiments	35
2.4. Prototype usage out of the lab	36
2.4.1. Example: A prototype in the real world.....	36
2.5. What can't FPGA-based prototyping do for us?	38
2.5.1. An FPGA-based prototype is not a simulator	38
2.5.2. An FPGA-based prototype is not ESL.....	39
2.5.3. Continuity is the key	39
2.6. Summary: So why use FPGA-based prototyping?	40
CHAPTER 3 FPGA technology today: chips and tools	41
3.1. FPGA device technology today	41
3.1.1. The Virtex [®] -6 family: an example of latest FPGAs	42
3.1.2. FPGA logic blocks.....	43
3.1.3. FPGA memory: LUT memory and block memory	46
3.1.4. FPGA DSP resources.....	47
3.1.5. FPGA clocking resources.....	49
3.1.6. FPGA input and output	51
3.1.7. Gigabit transceivers	53
3.1.8. Built-in IP (Ethernet, PCI Express [®] , CPU etc.)	54
3.1.9. System monitor.....	55
3.1.10. Summary of all FPGA resource types	56
3.2. FPGA-based Prototyping process overview	57
3.3. Implementation tools needed during prototyping	59
3.3.1. Synthesis tools	60
3.3.2. Mapping SoC design elements into FPGA	61
3.3.3. Synthesis and the three “laws” of prototyping	63
3.3.4. Gated clock mapping	65
3.4. Design partitioning flows	66
3.4.1. Pre-synthesis partitioning flow.....	67
3.4.2. Post-synthesis partitioning flow	68
3.4.3. Alternative netlist-based partitioning flow	70
3.4.4. Partitioning tool example: Certify [®]	72
3.5. FPGA back-end (place & route) flow	73

3.5.1. Controlling the back-end.....	75
3.5.2. Additional back-end tools	77
3.6. Debugging tools	77
3.6.1. Design instrumentation for probing and tracing.....	78
3.6.2. Real-time signal probing: test points	78
3.6.3. Real-time signal probing: non-embedded	80
3.6.4. Non real-time signal tracing.....	81
3.6.5. Signal tracing at netlist level.....	82
3.6.6. Signal tracing at RTL.....	85
3.6.7. Summarizing debugging tool options	89
3.7. Summary.....	90
CHAPTER 4 Getting started	91
4.1. A getting-started checklist	91
4.2. Estimating the required resources: FPGAs.....	92
4.2.1. How mature does the SoC design need to be?	93
4.2.2. How much of the design should be included?.....	94
4.2.3. Design blocks that map outside of the FPGA	95
4.2.4. How big is an FPGA?	97
4.2.5. How big is the whole SoC design in FPGA terms?.....	99
4.2.6. FPGA resource estimation	100
4.2.7. How fast will the prototype run?	102
4.3. How many FPGAs can be used in one prototype?	104
4.4. Estimating required resources.....	106
4.5. How long will it take to process the design?.....	106
4.5.1. Really, how long will it take to process the design?	108
4.5.2. A note on partitioning runtime	109
4.6. How much work will it be?	109
4.6.1. Initial implementation effort	110
4.6.2. Subsequent implementation effort.....	111
4.6.3. A note on engineering resources	111
4.7. FPGA platform	112
4.8. Summary.....	113
CHAPTER 5 Which platform? (1) build-your-own.....	115
5.1. What is the best shape for the platform?	115
5.1.1. Size and form factor.....	115

5.1.2. Modularity.....	117
5.1.3. Interconnect.....	119
5.1.4. Flexibility	121
5.2. Testability	122
5.3. On-board clock resources	123
5.3.1. Matching clock delays on and off board.....	124
5.3.2. Phase-locked loops (PLL).....	125
5.3.3. System clock generation	126
5.4. Clock control and configuration	128
5.5. On-board Voltage Domains.....	128
5.6. Power supply and distribution	129
5.6.1. Board-level power distribution.....	131
5.6.2. Power distribution physical design considerations.....	132
5.7. System reliability management.....	133
5.7.1. Power supply monitoring	133
5.7.2. Temperature monitoring and management	134
5.7.3. FPGA cooling	136
5.8. FPGA configuration	137
5.9. Signal integrity.....	138
5.10. Global start-up and reset.....	139
5.11. Robustness	139
5.12. Adopting a standard in-house platform.....	140
CHAPTER 6 Which platform? (2) ready-made.....	143
6.1. What do you need the board to do?.....	143
6.2. Choosing the board(s) to meet your goals	144
6.3. Flexibility: modularity.....	146
6.4. Flexibility: interconnect	147
6.5. What is the ideal interconnect topology?	150
6.6. Speed: the effect of interconnect delay	153
6.6.1. How important is interconnect flight time?	156
6.7. Speed: quality of design and layout	157
6.8. On-board support for signal multiplexing	158
6.9. Cost and robustness.....	159
6.9.1. Supply of FPGAs governs delivery of boards.....	160
6.10. Capacity	160

6.11. Summary	162
CHAPTER 7 Getting the design ready for the prototype	165
7.1. Why “get the design ready to prototype?”	165
7.1.1. RTL modifications for prototyping	166
7.2. Adapting the design’s top level	167
7.2.1. Handling the IO pads	168
7.2.2. Handling top-level chip support elements	168
7.3. Clock gating	170
7.3.1. Problems of clock gating in FPGA	171
7.3.2. Converting gated clocks	172
7.4. Automatic gated-clock conversion	174
7.4.1. Handling non-convertible gating logic	176
7.4.2. Clock gating summary	179
7.5. Selecting a subset of the design for prototyping	180
7.5.1. SoC block removal and its effect	180
7.5.2. SoC element tie-off with stubs	183
7.5.3. Minimizing and localizing RTL changes	184
7.5.4. SoC element replacement with equivalent RTL	186
7.5.5. SoC element replacement by inference	189
7.5.6. SoC element replacement by instantiation	191
7.5.7. Controlling inference using directives	193
7.6. Handling RAMs	194
7.7. Handling instantiated SoC RAM in FPGA	195
7.7.1. Note: RAMs in Virtex®-6 FPGAs	195
7.7.2. Using memory wrappers	197
7.7.3. Advanced self-checking wrappers	204
7.8. Implementing more complex RAMs	207
7.8.1. Example: implementing multiport RAMs	207
7.8.2. Example: bit-enabled RAMs	209
7.8.3. NOTE: using BlockRAM as ROMs	212
7.9. Design implementation: synthesis	212
7.9.1. Note: using existing constraints for the SoC design	213
7.9.2. Tuning constraints	215
7.10. Prototyping power-saving features	215
7.11. Design implementation: place & route	216

7.12. Revision control during prototyping	218
7.13. Summary.....	219
CHAPTER 8 Partitioning and reconnecting 221	
8.1. Do we always need to partition across FPGAs?	221
8.1.1. Do we always need EDA partitioning tools?	222
8.2. General partitioning overview	222
8.2.1. Recommended approach to partitioning	223
8.2.2. Describing board resources to the partitioner	224
8.2.3. Estimate area of each sub-block	225
8.2.4. Assign SoC top-level IO	226
8.2.5. Assign highly connected blocks	227
8.2.6. Assign largest blocks	229
8.2.7. Assign remaining blocks	231
8.2.8. Replicate blocks to save IO.....	231
8.2.9. Multiplex excessive FPGA interconnect	234
8.2.10. Assign traces.....	234
8.2.11. Iterate partitioning to improve speed and fit	237
8.3. Automated partitioning.....	239
8.4. Improving prototype performance	240
8.4.1. Time budgeting at sequential boundaries	241
8.4.2. Time budgeting at combinatorial boundaries.....	242
8.5. Design synchronization across multiple FPGAs	244
8.5.1. Multi-FPGA clock synchronization.....	244
8.5.2. Multi-FPGA reset synchronization.....	247
8.5.3. Multi FPGA start-up synchronization	250
8.6. More about multiplexing	251
8.6.1. What do we need for inter-FPGA multiplexing?	251
8.7. Multiplexing schemes	253
8.7.1. Schemes based on multiplexer	253
8.7.2. Note: qualification criteria for multiplexing nets	254
8.7.3. Schemes based on shift-registers.....	255
8.7.4. Worked example of multiplexing	256
8.7.5. Scheme based on LVDS and IOSERDES.....	262
8.7.6. Which multiplexing scheme is best for our design?.....	264
8.8. Timing constraints for multiplexing schemes	265

8.9. Partitioning and reconnection: summary.....	266
CHAPTER 9 Design-for-Prototyping	267
9.1. What is Design-for-Prototyping?.....	267
9.1.1. What's good for FPGA is usually good for SoC.....	268
9.2. Procedural guidelines	268
9.2.1. Integrate RTL team and prototypers.....	269
9.2.2. Define list of deliverables for prototyping team	270
9.2.3. Prototypers work with software team	272
9.3. Integrate the prototype with the verification plan.....	272
9.3.2. Documentation well and use revision control.....	274
9.3.3. Adopt company-wide standard for hardware.....	274
9.3.4. Include Design-for-Prototyping in RTL standards.....	274
9.4. Design guidelines.....	275
9.4.1. Follow modular design principles	277
9.4.2. Pre-empt RTL changes with 'define and macros	278
9.4.3. Avoid latches.....	279
9.4.4. Avoid long combinatorial paths	279
9.4.5. Avoid combinatorial loops.....	280
9.4.6. Provide facility to override FFs with constants.....	280
9.5. Guidelines for isolating target specificity	281
9.5.1. Write pure RTL code	281
9.5.2. Make source changes as low-impact as possible.....	281
9.5.3. Maintain memory compatibility	282
9.5.4. Isolation of RAM and other macros	282
9.5.5. Use only IP that has an FPGA version or test chip	284
9.6. Clocking and architectural guidelines	284
9.6.1. Keep clock logic in its own top-level block.....	285
9.6.2. Simplify clock networks for FPGA	285
9.6.3. Design synchronously.....	286
9.6.4. Synchronize resets	286
9.6.5. Synchronize block boundaries.....	286
9.6.6. Think how the design might run if clocked slowly	287
9.6.7. Enable bottom-up design flows	287
9.7. Summary.....	288
CHAPTER 10 IP and high-speed interfaces.....	289

10.1. IP and prototyping	289
10.2. IP in many forms.....	290
10.2.1. IP as RTL source code	291
10.2.2. What if the RTL is not available?.....	291
10.2.3. IP as encrypted source code	292
10.2.4. Encrypted FPGA netlists.....	293
10.2.5. Encrypted FPGA bitstreams.....	293
10.2.6. Test chips	295
10.2.7. Extra FPGA pins needed to link to test chips.....	297
10.3. Soft IP.....	298
10.3.1. Replacing instantiated soft IP.....	301
10.3.2. Replacing inferred soft IP	301
10.3.3. Replacing synthetic soft IP.....	302
10.3.4. Other FPGA replacements for SoC soft IP	304
10.4. Peripheral IP	304
10.4.1. Use mode 1: prototype the IP itself	305
10.4.2. Use mode 2: prototype the IP as part of an SoC.....	306
10.4.3. Use mode 3: prototype IP for software validation.....	307
10.5. Use of external hard IP during prototyping	308
10.6. Replacing IP or omitted structures with FPGA IP.....	308
10.6.1. External peripheral IP example: PCIe and SATA.....	309
10.6.2. Note: speed issues, min-speed.....	310
10.7. Summary.....	311
CHAPTER 11 Bring up and debug: the prototype in the lab.....	313
11.1. Bring-up and debug—two separate steps?	313
11.2. Starting point: a fault-free board.....	314
11.3. Running test designs.....	316
11.3.1. Filter test design for multiple FPGAs	317
11.3.2. Building a library of bring-up test designs.....	319
11.4. Ready to go on board?.....	320
11.4.1. Reuse the SoC verification environment	321
11.4.2. Common FPGA implementation issues.....	321
11.4.3. Timing violations.....	322
11.4.4. Improper inter-FPGA connectivity.....	325
11.4.5. Improper connectivity to the outside world	327

11.4.6. Incorrect FPGA IO pad configuration	328
11.5. Introducing the design onto the board.....	331
11.5.1. Note: incorrect startup state for multiple FPGAs	332
11.6. Debugging on-board issues	333
11.6.1. Sources of faults	333
11.6.2. Logical design issues	334
11.6.3. Logic debug visibility	335
11.6.4. Bus-based design access and instrumentation.....	336
11.6.5. Benefits of a bus-based access system.....	339
11.6.6. Custom debug using an embedded CPU.....	341
11.7. Note: use different techniques during debug	342
11.8. Other general tips for debug	343
11.9. Quick turn-around after fixing bugs.....	346
11.9.1. Incremental synthesis flow.....	347
11.9.2. Automation and parallel synthesis.....	349
11.9.3. Incremental place & route flow	350
11.9.4. Combined incremental synthesis and P&R flow	351
11.10. A bring-up and debug checklist	352
11.11. Summary.....	353
CHAPTER 12 Breaking out of the lab: the prototype in the field.....	355
12.1. The uses and benefits of a portable prototype	355
12.2. Planning for portability	357
12.2.1. Main board physical stiffness.....	358
12.2.2. Daughter board mounting	358
12.2.3. Board mounting holes	358
12.2.4. Main board connectors.....	359
12.2.5. Enclosure.....	359
12.2.6. Cooling.....	360
12.2.7. Look and feel.....	361
12.2.8. Summary	362
CHAPTER 13 Prototyping + Verification = The Best of Both Worlds	363
13.1. System prototypes	363
13.2. Required effort.....	364
13.3. Hybrid verification scenarios.....	365
13.4. Verification interfaces	365

13.4.1. Interfaces for co-simulation	366
13.4.2. Interfaces for transaction-based verification	368
13.4.3. TLMs and transactors	369
13.4.4. SCE-MI	369
13.4.5. SCE-MI 2.0 implementation example	371
13.4.6. VMM HAL.....	373
13.4.7. Physical interfaces for co-verification	375
13.5. Comparing verification interface technologies.....	375
13.6. Use models – more detail	377
13.6.1. Virtual platform re-using existing RTL	377
13.7. Virtual platform for software.....	378
13.7.1. Virtual platform as a testbench.....	379
13.7.2. Virtual and physical IO (system IO).....	380
13.7.3. Virtual ICE	381
13.8. System partitioning	383
13.9. Case study: USB OTG	384
13.9.1. USB OTG System overview	384
13.9.2. Integration use models	385
13.9.3. Innovator and VCS	385
13.9.4. Innovator and CHIPit or HAPS.....	386
13.9.5. Virtual platform	387
CHAPTER 14 The future of prototyping	391
14.1. If prediction were easy. . ..	391
14.2. Application specificity	391
14.3. Prototyping future: mobile wireless and consumer.....	392
14.4. Prototyping future: networking.....	394
14.5. Prototyping future: automotive	397
14.6. Summary: software-driven hardware development	400
14.7. Future semiconductor trends.....	401
14.8. The FPGA's future as a prototyping platform.....	402
14.9. Summary.....	403
CHAPTER 15 Conclusions	405
15.1. The FPMM approach to FPGA-based prototyping.....	405
15.2. SoCs are larger than FPGAs	406
15.3. SoCs are faster than FPGAs	407

15.4. SoCs designs are FPGA-hostile	407
15.5. Design-for-Prototyping beats the three laws	408
15.6. So, what did we learn?	409
APPENDIX A: Worked Example: Texas Instruments.....	411
A1. Design background: packet processing sub-system.....	411
A2. Why does Texas Instruments do prototyping?	412
A3. Testing the design using an FPGA-based prototype	413
A4. Implementation details	415
A5. High-speed scenario	416
A6. Low-speed scenario.....	417
A7. Interesting challenges	418
A8. Summary of results	421
APPENDIX B: Economics of making prototype boards.....	421
B1. Prototyping hardware: should we make or buy?.....	423
B2. Cost: what is the total cost of a prototyping board?.....	424
B3. Direct cost: personnel	424
B4. Direct cost: equipment and expertise	425
B5. Direct cost: material and components	427
B6. Direct cost: yield and wastage	429
B7. Direct cost: support and documentation	429
B8. Business cost: time	430
B9. Business cost: risk	433
B10. Business cost: opportunity	435
B11. CCS worked example results	435
B12. Summary	437
References and Bibliography	438
Acknowledgements	443
Figure Sources	445
Glossary of key terms	447
Index	451
About the authors	469
About Synopsys Press.....	470

This chapter will establish some definitions and outline the challenges we are trying to overcome with FPGA-based prototyping. We will explore the complexity of SoC-based systems and the challenges in their verification. We will also compare and contrast FPGA-based prototyping with other prototyping methods including system-level virtual modeling. After this chapter we will be ready to dive deeper into the ways that FPGA-based Prototyping has benefited some real projects and give some guidance on what is possible with FPGA-based prototyping technology.

1.1. Moore was right!

Since Gordon E. Moore described the trend of how many transistors can be placed inexpensively on an integrated circuit, electronic design enabled by semiconductor design has grown at a hard-to-imagine pace. The trend of transistors doubling every two years has already continued for more than half a century and is not expected to stop for a while despite repeated predictions that it would end soon.

A detailed review of the major trends driving chip design later in this chapter will make it clear why prototyping has grown in adoption, and is even considered mandatory in many companies. To further understand this trend, typical project dynamics and their effort distributions need to be understood.

1.1.1. SoC: A definition . . . for this book at least

Let's start with a definition. For the purposes of this book, we define system on chip (SoC) as a device which is designed and fabricated for a specific purpose, for exclusive use by a specific owner. Some might think of SoC as a particular form of an ASIC (application specific integrated circuit) and they would be correct, but for the purposes of this book, we will refer only to SoCs. We will stick to the definition that an SoC always includes at least one CPU and runs embedded software. In comparison, an ASIC does not necessarily include a CPU and to that extent, SoCs can be considered to be a superset of ASICs.

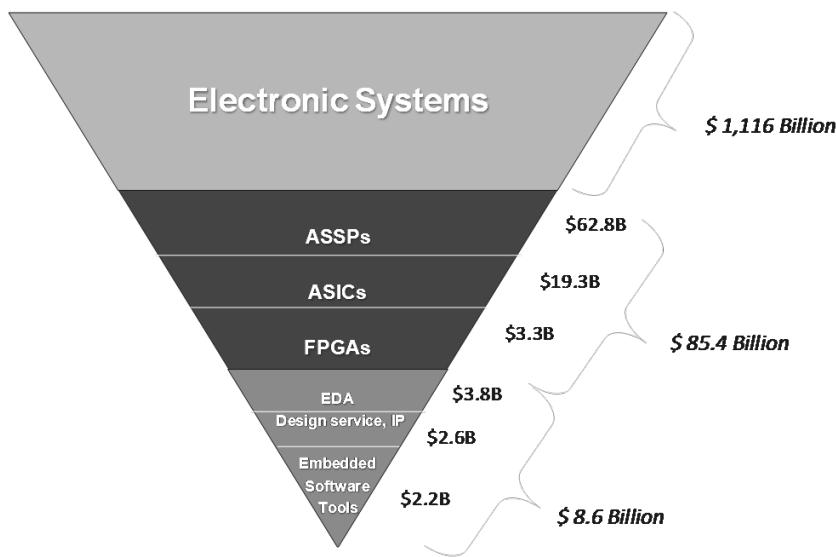
We do not mean to imply that those who are creating an ASIC, ASSP (application-specific silicon product) or silicon produced by COT (customer's own tooling) or by third-party foundries should read another book. Technologies are not mutually unique for FPGA-based prototyping purposes and in fact, many FPGA-based prototyping projects are of an ASSP or even just pieces of IP that may be used in many different SoC designs.

As far as FPGA-based prototyping is concerned, if it works for SoC then it will work for any of the above device types. The reason for this book's focus on SoC is that the greatest value of FPGA-based prototyping is in its unique ability to provide a fast and accurate model of the SoC in order to allow validation of the software.

1.2. The economics of SoC design

SoC designs are all around us. We can find them in all the new headline-grabbing consumer electronics products as well as in the most obscure corners of pure research projects in places like CERN, and in the guidance systems of interstellar probes.

Figure 1: The relationship of IC design to the electronics market

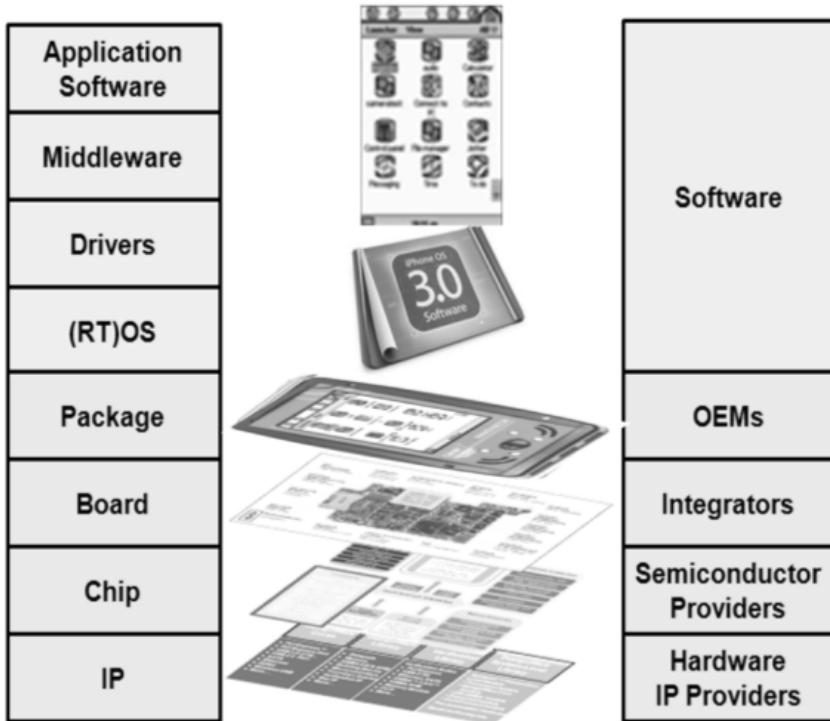


For consumer products in particular, there is a seemingly insatiable hunger for maximum intelligence and functionalities in devices such as smart phones, cameras or portable media players. To meet these requirements a typical SoC design will

include several microprocessors, one or more digital signal processors and some different interfaces such as Bluetooth™ or WLAN, high resolution graphics and so on. That all adds up to a lot of software.

Considering the IC development and manufacture as a whole, it appears in Figure 1 as an inverted triangle. The figures shown are for 2009, and we see that chip development was a market of about \$85.4 billion and was enabled by a \$8.6 billion market for EDA tools, design services, IP and embedded software tools. Supported by this semiconductor design and manufacture is a huge \$1.116 billion market for electronic systems, which contain for example all the consumer gadgets, wireless devices and electronics we crave as end consumers.

Figure 2: Hardware-software teardown of a consumer device



EDA tools, which include various types of prototyping for different stages within a design, are recently focusing to specifically enabling the design chain from IP providers, semiconductor providers, integrators and OEMs. Prototyping plays a key role in those interactions as early prototypes enable communication of requirements from customers to suppliers and early software development and verification for customers from suppliers.

To understand the impact that prototyping in its many forms can achieve, let's consider a typical complex SoC design. Figure 2 shows the tear down of a typical smartphone. The end user experience is largely influenced by the applications with which they are presented. Good old hardware, analog and antenna design is obviously still important but the user only really notices them when they go wrong! User applications are enabled by a software stack of middleware, operating system and drivers; all of which are specifically designed to make the software as independent of the hardware as possible.

For example, application developers do not have direct access to the device's hardware memory, timing or other low-level hardware aspects. The stack of software is matched by a stack of hardware elements. The end device uses several boards, comprised of several peripherals and chips, which contain various blocks, either reused as IP or specifically developed by chip providers to differentiate their hardware.

The dependencies of hardware and software result in an intricate relationship between different company types. IP providers sell to semiconductor providers, who sell to integrators who sell to OEMs, all of whom are enabling software developers. Enablement of these interactions has arguably become the biggest problem to be addressed by tool vendors today.

The main challenges for this enablement have become today:

- (a) The enablement of software development at the earliest possible time.
- (b) Validation of hardware / software in the context of the target system.
- (c) Design and reuse of the basic building blocks for chips
 - processors
 - accelerators
 - peripherals
 - interconnect fabrics (e.g., ARM AMBA® interconnect)
- (d) Architecture design of the chips assembled from the basic building blocks.

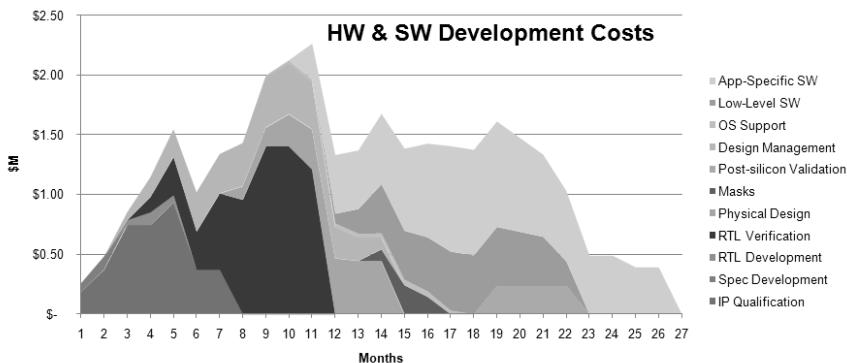
1.2.1. Case study: a typical SoC development project

Considering the bottom three layers of the hardware stack in Figure 2, let's analyze a specific chip development project and the potential impact of prototyping. The chosen example is a wireless headset design by a large semiconductor company, performed in a mainstream 65nm technology. The chip is targeted for a high volume, fast moving market and has an expected production run of 27 months with an average volume of 1.5 million units per month and average selling price of

\$5.50. Things go well during development and only one metal mask spin is required allowing six months of customer and field evaluations after first silicon is available. In total the development cost for the project is estimated as \$31,650,000 based on a development cost model described in an International Business Systems study with scaling factors for mainstream applications.

Let's now consider Figure 3, which illustrates how chip development cost is spread over the typical 12-month hardware design cycle, from complete specification to final verified RTL, ready for layout. Indeed, RTL verification consumes the majority of the effort and is the critical element in determining the project length of 12 months. Another portion of the design with significant impact is the overall design management accompanying the actual development of code. Physical design is finished about 15 months into the project (i.e., three months after RTL is verified) and then masks are prepared by month 17. The post silicon validation ramps up with engineering samples available in month 19 and takes several months.

Figure 3: Project effort for a 65nm wireless headset design



As Figure 3 further illustrates, software development ramps up in this project when RTL is largely verified and stable. It is split here between OS support and porting, low-level software development and high-level application software development. All the software development effort here is still the responsibility of the chip provider, rather than third-party providers. Overall, software development consumes 40% of the total cost for this design and extends the project schedule to a total of 27 months.

When amortizing development and production cost onto expected sales, this project reaches break even after about 34 months, i.e., seven months after product launch but almost three years after starting product development. The challenge in this example is that we have to predict nearly three years in advance what is going to sell in high-volumes in order to specify our chip. How can this almost intolerable situation be made easier? The answer is to “start software sooner.”

Using the calculator for return on investment (ROI) developed by the Global Semiconductor Association (GSA), it can be calculated that if software development and validation started seven months earlier in our example project, production could have started three months earlier and subsequently the time to break even would have been reduced by five months. In addition a \$50 million revenue gain could have been expected over the production volume due to extra first-to-market design-wins for the chip.

This is exactly what prototyping in its many forms can achieve. The earlier start to software development and validation provided by prototyping means that its impact on ROI can be very significant.

For a deeper understanding of requirements and benefits of prototyping, let's look at the different types of prototyping available today from virtual to FPGA-based.

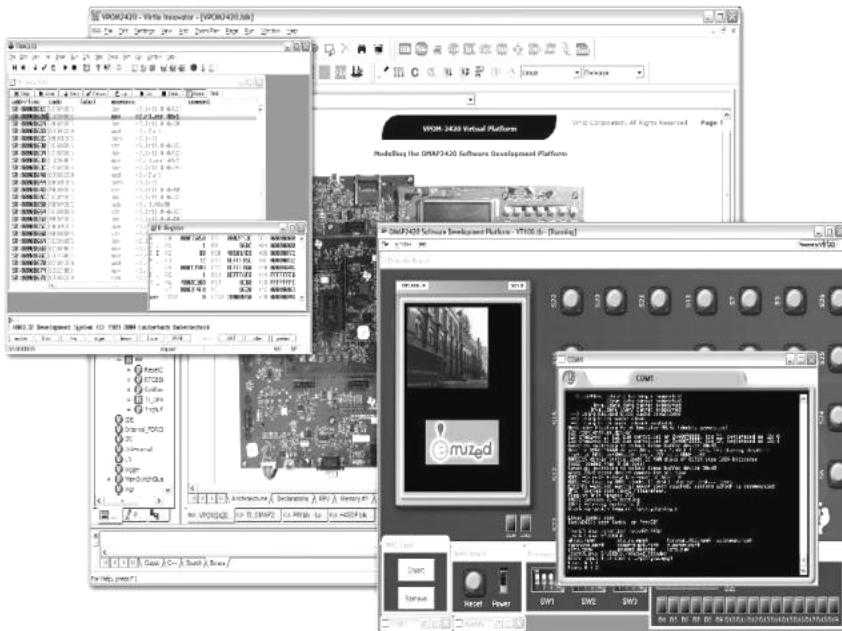
1.3. Virtual platforms: prototyping without hardware

There are various forms of prototyping which we can employ in our projects. Available earliest in a project are virtual prototypes. They represent fully functional but loosely-timed software models of SoCs, boards, virtualized IOs and user interfaces, all running on a host-based simulation. Virtual prototypes can execute unmodified production software code on processor instruction set simulators and they run close to real-time. Being fully host-based virtual prototypes they can also offer good system visibility and control, which is especially useful for debug on multicore CPUs. Virtual prototypes can also have virtual user interfaces, allowing real-time interaction with us slow humans. The screenshot excerpts shown in Figure 4 are from a virtual prototype of an OMAP design running on the Synopsys Innovator tool. Here we see not only recognizable simulation windows but also the representation of key controls on the board and input from a virtual camera, in this case linked to a webcam on the host PC running the simulation, all done without hardware. We shall come back to Innovator in chapter 13 when we discuss the linking of an FPGA-based prototype with a virtual simulation.

While virtual prototypes offer very high speed (multiple tens of MIPS) when using loosely-timed models they do not offer the timing accuracy preferred by hardware design teams. More timing-accurate software models can be added to a virtual prototype but then their simulation speed will degrade to the single-digit MIPS range or even lower depending on the mix of cycle-accurate versus loosely-timed models.

However, virtual prototypes are available earliest in the flow, assuming models are available, so they are perfect for early software debugging. Virtual prototypes provide almost complete insight into the behavior of the system and they are also easy to replicate for multiple users.

Figure 4: Pre-Silicon virtual prototype of OMAP design



Finally, because they are created before the RTL, virtual prototypes allow co-development of hardware architecture along with early software. For example, extra or different CPUs might be added if the virtual prototype shows that there is not enough processing bandwidth for the product's concurrent applications.

1.3.1. SDK: a very common prototyping environment

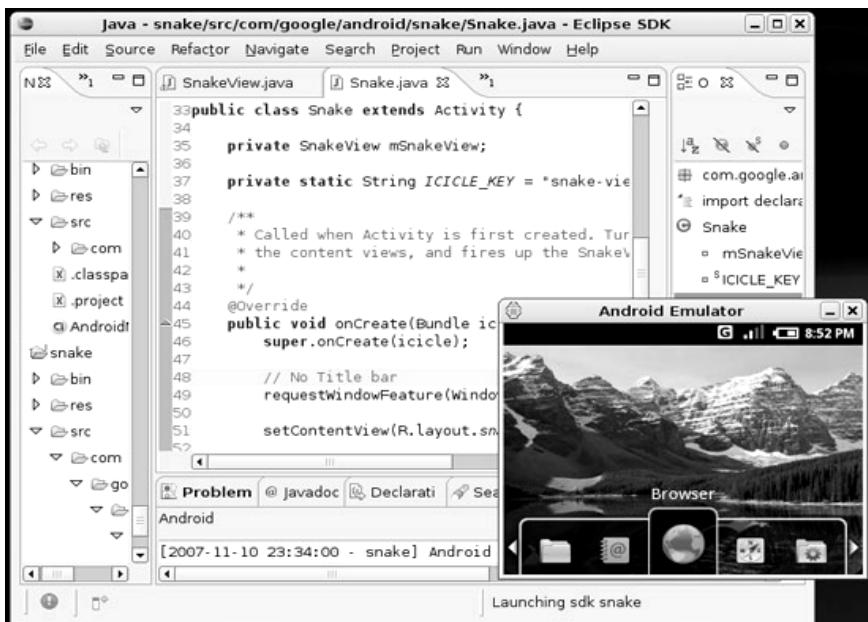
Related to virtual prototypes are so-called software development kits, or SDKs, of which a very common example is the SDK for developing applications for the Apple iPhone®. This SDK was downloaded more than 100,000 times in the first couple of days of its availability and so we can consider this a very widely available prototyping platform. Although simple in comparison, SDKs offer many of the advantages of full virtual prototypes, however, their accuracy is often more limited because they may not represent the actual registers as accurately as virtual prototypes.

Their aim is to have “just enough accuracy” in order to fool the application into thinking that it is running on the final platform. SDKs allow programming and interfacing over a higher-level application programming interface (API) into the platform. The developed software is usually compiled for the host machine on

which the SDK executes and then requires re-compilation to the actual target processor after programmers have verified functionality on the SDK.

Figure 5 shows a screenshot of an Android™ SDK. Programming of applications is done in high-level languages like C using higher-level operating system APIs. The programming is done completely independent of the actual target hardware, so when actually targeting the final device, recompilation is necessary. User interfaces of the target device – as shown in Figure 5, can be modeled so that the end-user environment can be experienced virtually.

Figure 5: Android™ SDK screenshot showing GUI emulation



1.3.2. FPGA: prototyping in silicon . . . but pre-silicon

Available later in the design flow, but still well before real silicon, an FPGA-based prototype also serves as a vehicle for software development and validation. FPGA-based prototypes are fully functional hardware representations of SoCs, boards and IOs. Because they implement the same RTL code as the SoC and run at almost real-time speed with all external interfaces and stimulus connected, they are very accurate. They offer higher system visibility and control than the actual silicon will provide when available but do not match the debug and control capabilities of

virtual platforms, or any other simulator, so they are not the first platforms we would choose on which to debug all of our RTL.

The key advantage of FPGA-based prototypes is their ability to run at high speed, yielding tens of MIPS per modeled CPU, while maintaining RTL accuracy. They are typically available later in the design flow than virtual prototypes because RTL needs to be available and relatively mature. Due to the complexity and effort of mapping the RTL to FPGA-based prototypes, it is not really feasible to use them before RTL verification has stabilized. For the same reason, FPGA-based prototypes are not intended for use as hardware/software co-development platforms because at this point in the SoC design flow, the hardware (i.e., the RTL) is pretty much fixed and partially verified. Design teams will be very hesitant to change the hardware architecture by the time the FPGA-based prototype is running, unless some major architectural bottlenecks have been discovered.

Finally, once stable and available, the cost of replication and delivery for FPGA-based prototypes is higher than for software-based virtual platforms, however still considerably cheaper than emulators, which we shall discuss next.

1.3.3. Emulators: prototyping or verification?

Emulation provides another hardware-based alternative to enable software development but differs from FPGA-based prototyping in that it aims at lower performance but with more automation. Emulators have more automated mapping of RTL into the hardware together with faster compile times, but the execution speed will be lower and typically drop to below the single-MIPS level. The cost of emulation is also often seen as a deterrent to replicating it easily for software development despite the fact that emulators are popular with software engineers because of their ease of use.

As with FPGA-based prototypes, emulators are not realistic platforms for hardware-software co-development because they require the RTL to be available. A more likely use for emulators is as an accelerator for normal RTL simulation and so many consider emulation not as a prototyping platform but as an extension to the normal verification environment; a kind of go-faster simulator. An emulator can actually be used for software development, however, only when the software needs cycle-accuracy and high-visibility into the RTL and can tolerate very slow run speed. Software would need to be limited to short duration runs, such as the boot ROM code, because the slow running speed will mean that runtimes can be very long; certainly too long for long software tasks, such as user applications or OS bring-up.

1.3.4. First silicon as a prototype platform

Finally, after the actual silicon is available, early prototype boards using first silicon samples can enable software development on the actual silicon. Once the chip is in production, very low-cost development boards can be made available. At this point, the prototype will run at real-time speed and full accuracy. Software debug is typically achieved with specific hardware connectors using the JTAG interface and connections to standard software debuggers. While prototype boards using the actual silicon are probably the lowest-cost option, they are available very late in the design flow and allow almost no head start on software development. In addition, the control and debug insight into hardware prototypes is very limited unless specific on-chip instrumentation (OCI) capabilities are made available. In comparison to virtual prototypes, they are also more difficult to replicate – it is much easier to provide a virtual platform for download via the internet than to ship a board and deal with customs, bring-up and potential damages to the physical hardware.

As can be seen by this overview, prototyping is focused on providing early representations of hardware, specifically of chips and their surrounding peripherals. Prototypes are applicable for different use models, which in exchange have an impact on requirements.

1.4. Prototyping use models

As indicated earlier, prototyping is done today using different execution engines. Once a chip development project has started, project managers are asked almost immediately to provide early representations – prototypes – of the “chip-to-be” for various purposes, such as:

- Marketing needs material and basic documentation to interact with early adopters.
- Software developers would like executable representations of the design under development to allow them to start porting operating systems
- Hardware developers would also like executable specifications to validate that their implementations are correct.
- Prototypes are in high demand from day one! The need is driven by three main use models: architecture exploration, software development and verification.

1.4.1. Prototyping for architecture exploration

At the beginning of a project, architecture exploration allows chip architects to make basic decisions with respect to the chip topology, performance, power consumption and on-chip communication structures. For example, information gathered early on cache utilization, performance of processors, bus bandwidth, burst rates and memory utilization drives basic architecture decisions.

In an ideal world, chip architects would love to get prototypes with fully accurate models – representing all internals of the design – while running at full speed. Unfortunately, both characteristics typically cannot be achieved by the same model. Fully accurate data cannot be gathered until the final chip comes back from fabrication or at least until late in the design cycle, when RTL is available and verified. At that point, FPGA prototypes can be used to execute the design close to real time.

Chip architects also interact with “early adopter” customers and ideally would like executable specifications to demonstrate key features of the design. However, in reality, chip architects mostly rely on tools like Microsoft Excel™ for basic, static architectural analysis. Often they have to rely on their experience and back of the envelope assessments. As a result, interaction with early adopter customers happens based on written specifications and lots of joint discussions at white boards.

1.4.2. Prototyping for software development

Software developers would ideally like to start their porting of legacy code and development of new software functionality from the get go, i.e., when the hardware development kicks off. They would like to receive an executable representation of the chip, which runs at real-time and accurately reflects all the software related interfaces with the hardware (like register images). Depending on the type of software being developed, users may require different accuracy from the underlying prototype. The type of software to be developed directly determines the requirements regarding how accurately hardware needs to be executed:

- Application software can often be developed without taking the actual target hardware accuracy into account. This is the main premise of SDKs, which allow programming against high-level APIs representing the hardware.
- For middleware and drivers, some representation of timing may be required. For basic cases of performance analysis, timing annotation to caches and memory management units may be sufficient, as they are often more important than static timing of instructions when it comes to performance.

- For real-time software, high-level cycle timing of instructions can be important in combination with micro-architectural effects.
- For time-critical software – for example, the exact response behavior of interrupt service routines (ISRs) – fully cycle-accurate representations are preferred.

Often still today, developers will start software development “blindly,” based on register specifications, but are then plagued by not being in sync with changes the hardware team still may make to the register specifications. For derivative product families, application software is often developed using high-level APIs, which can be executed on a previous generation chip. The underlying drivers, OS and middleware, which become available later, make sure that the APIs remain the same and do not break legacy software.

1.4.3. Prototyping for verification

Early on, the chip environment is represented using traces and traffic generators. Early test benches essentially define the use model scenarios for the chip under development.

In a typical design, just as many bugs hide in the test bench as in the actual design itself. Therefore, it is important to start with the development of test benches for verification as early as possible. Ideally, verification engineers would like an executable representation of the “device under test” (DUT) to be available from the start. Similar to software development’s need for different models, verification has requirements for different levels of accuracy as well.

High level models of the DUT will enable development of verification scenarios. Models of the DUT with accurate registers and pure functional representation of the DUT’s behavior satisfy a fair share of test bench development. For verification of timing and detailed pipeline latencies, timing approximation may initially be sufficient but eventually a cycle-accurate representation at the register transfer level (RTL) will be required.

An important trend in hardware verification is the move of functional verification into software, which executes on processors embedded in the design. In answer to the recent survey question “Do you use software running on the embedded processors in your design for verification of the surrounding hardware?”, more than 50% of the respondents answered that they are already using embedded software for verification, and one in ten of them also use it with a focus on post-silicon validation.

The advantage of this CPU-based approach is verification reuse:

- Tests are executed on processors and developed initially using fast instruction-accurate processor models interacting with transaction-level models (TLM) of the DUT, accessing it through its register interface.
- Later the tests can be reused in mixed TLM/RTL simulations as well as in pure RTL simulations for which the processor is mapped into RTL.
- The tests can still be used for hardware prototypes with the processor executing as TLM on the workstation, connected to the hardware via high-speed TLM interfaces.
- Processors executing the tests can also run as RTL in the FPGA prototype or can be brought as chip on a plug-in board into the FPGA prototype.
- Finally, when the chip is back from fabrication, software-based tests can be used for post-silicon verification as well.

1.5. User priorities in prototyping

With all of those trends combined, prototyping of chips is becoming a clear requirement for successful chip design. However, different user priorities lead towards different prototyping options as the best solution. We can list them in a number of ways but in our case, we chose to highlight twelve different priorities, as listed below.

- **Time of availability:** once the specifications for our design are frozen, delays in delivery of our software validation environment directly impacts how quickly we can start and progress in the software part of our SoC project.
- **Execution speed:** ideally the chosen development method provides an accurate representation of how fast the real hardware will execute. For software regressions, execution that is faster than real-time can be beneficial.
- **Accuracy:** the type of software being developed determines how accurate the development methods have to be in order to represent the actual target hardware, ensuring that issues identified at the hardware/software boundary are not introduced by the development method itself.
- **Capacity:** can the prototype handle the largest SoC designs or is it not required to do so? How does performance and cost alter with increased design size? Can the platform be upgraded for larger designs in the future?
- **Development cost:** the cost of a development method is comprised of both the actual cost of production, as well as the overhead cost of bringing up hardware/software designs within it. The production cost determines how

easy a development method can be replicated to furnish software development teams.

- **Bring-up cost:** any required activity needed to enable a development method outside of what is absolute necessary to get to silicon can be considered overhead. Often the intensity of the pressure that software teams face to get access to early representations of the hardware determines whether or not the investment in bring-up cost is considered in order to create positive returns.
- **Deployment cost:** if we are to create multiple copies of our prototype then we need to be aware of how much each will cost to create, deploy, maintain and support in the labs of our end users and beyond.
- **Debug insight:** the ability to analyze the inside of a design, i.e., being able to access signals, registers and the state of the hardware/software design.
- **Execution control:** during debug, it is important to stop the representation of the target hardware using assertions in the hardware or breakpoints in the software, especially for designs with multiple processors in which all components have to stop in a synchronized fashion.
- **System interfaces:** if the target design is an SoC, it is important to be able to connect the design under development to real-world interfaces. For example, if a USB interface is involved, the software will need to connect to the real USB protocol stacks. Similarly, for network and wireless interfaces, connection to real-world software is a priority.
- **Turnaround time:** from a new set of source files, be they SystemC™ models or raw RTL, how long does it take to create a new version of the prototype? Is it measured in minutes, hours, days or weeks and what is required for the project in any case?
- **Value links to low power and verification:** prototypes do not have to be stand-alone platforms and it may add value if they can be linked to other parts of the SoC design team, particularly for verification. Prototyping before and after insertion of various implementation steps, for example modifications to reduce power, would also be valuable.

Probably no user has ever cared about all of these decision criteria at the same time and for any given SoC project, some will override others. We shall revisit most of these criteria as we progress through this book.

At the end of the book we will look to the future of prototyping as a whole and the place of FPGA-based prototyping in that future. On looking to the future we need to be aware of the recent past and trends that are emerging within the SoC user-base and wider industry. Let us look at those trends now.

1.6. Chip design trends

We will want to use our prototyping environment for future SoC projects but what will those projects look like? Understanding the eight major trends for chip design will lead to better readiness for future projects and a more flexible in-house prototyping methodology.

The eight major trends driving requirements for semiconductor design are:

- Further miniaturization towards smaller technology nodes.
- A decrease in overall design starts.
- Programmability combined with a rapid increase of embedded software content.
- IP reuse.
- Application specificity.
- Adoption of multicore architectures.
- Low power.
- An increase in the analog/mixed signal portion of chips.

All of these have profound impact on prototyping requirements and we shall quickly look at each and the supporting trend data in turn.

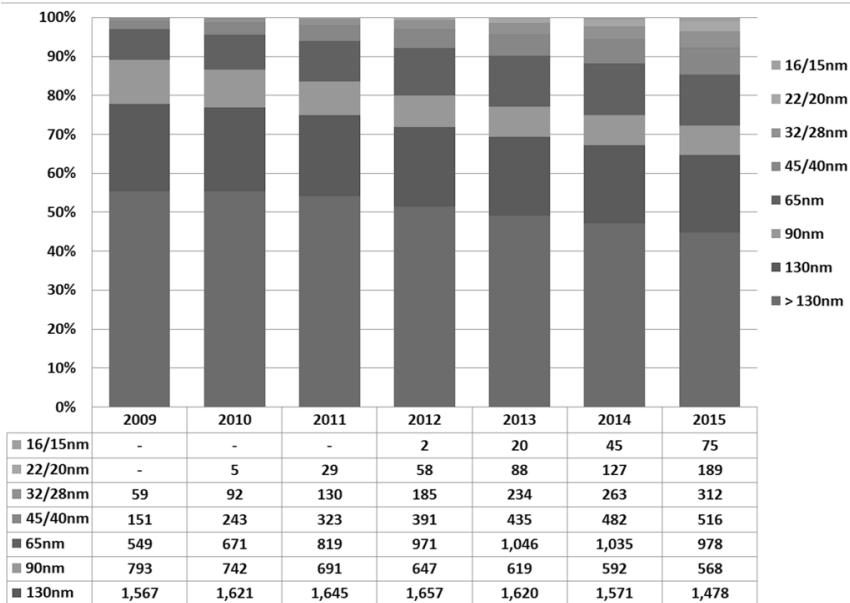
1.6.1. Miniaturization towards smaller technology nodes

In processor design the number of transistors has increased from 29,000 transistors defining the X86 in 1979 to 1.72 billion transistors by 2005 defining the Dual Core Itanium. That was an almost 60,000 fold increase over the time frame of 26 years. This trend has continued since and is likely to continue in the future and the number of design starts at smaller technology nodes will increase as outlined in Figure 6. This diagram (courtesy of the industry analyst, International Business Strategies Inc. (IBS) of Los Gatos, CA) shows each node as a percentage of all ASIC and SoC design starts.

Designs at the 65nm and 45nm nodes started in 2007 and have now become mainstream. As a result the size of designs to be prototyped has steadily increased, requiring more and more capacity for both software- and hardware-based prototypes.

Software-based prototypes are naturally limited in speed by the traditional serial execution of software. This has further increased the pressure to improve the speed of software simulation, especially for processor models.

Figure 6: Design starts per technology node (Source: IBS)



While fast simulation models have been available since the late 1990s using proprietary techniques, standardization has now enabled the combination of models from various sources into SystemC-based simulations without significant speed degradation using the open TLM-2.0 APIs.

For hardware-based prototypes this trend has further increased the pressure to adopt higher-density FPGAs for prototyping. Given that the capacity of FPGA prototyping is limited by the capacities of the available FPGAs, the only alternative is to divide and conquer and to only prototype smaller parts of the designs. To address this situation, FPGA prototypes have become more scalable using standard interfaces for stacking and expansion. Finally, in situations in which a prototype is to be tested within the system context of the chip under development, partitioning of the design can be difficult. Given that the number of FPGAs per prototyping board will increase to allow for sufficient capacity, the requirements on automatically partitioning the design across FPGAs has also increased.

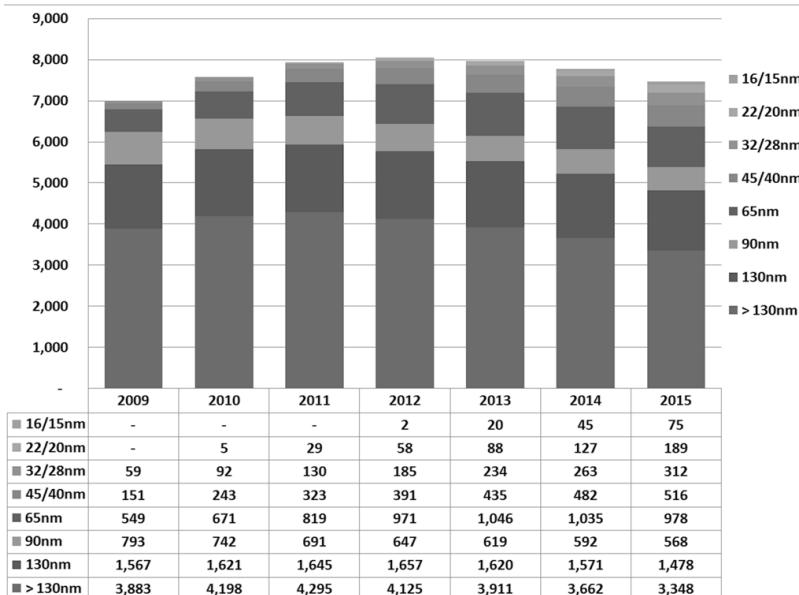
1.6.2. Decrease in overall design starts

On the flip side of the trend towards miniaturization is the reduced number of design starts. As indicated in Figure 7, the overall number of design starts for SoCs

is expected to decrease significantly. On first sight the development cost of modern designs is simply so high that fewer companies can afford SoC development.

However, in reality, the biggest drop will be in design starts for the older technologies, i.e., 130nm and above. Design teams will continue to design for leading-edge processes where software content, and the need for prototyping, is greatest.

Figure 7: Overall number of design starts per year (Source: IBS)



As a direct result of this trend the risk per design increases dramatically and more and more companies are already mandating prototyping of their designs prior to tape out to verify correctness and to avoid expensive re-spins. Prototyping can happen at various points in the design flow using a variety of different techniques. Overall, the decrease in design starts will only increase the risk per project even further and as a result prototyping will become even more important.

1.6.3. Increased programmability and software

The vast majority of electronic systems and products now include some element of programmability, which is in essence deferred functionality, which comes in several forms. First of all, estimates of the relative number of design starts for ASIC, ASPP and FPGAs show that the overwhelming number of design starts are in FPGAs and other Programmable Logic Devices; this clearly counts a programmable hardware.

Second, the number of FPGA design starts that include microprocessor units is growing very fast as well. This adds software programmability to programmable hardware.

In addition, a fair percentage of the ASIC and ASSP design starts contain embedded processors as well. As a result, the software adds programmability even dedicated SoC chips. As a result software is gaining significantly in importance, even in SoC projects.

Figure 8: Software effort as percentage of R&D expense (Source: IBS)

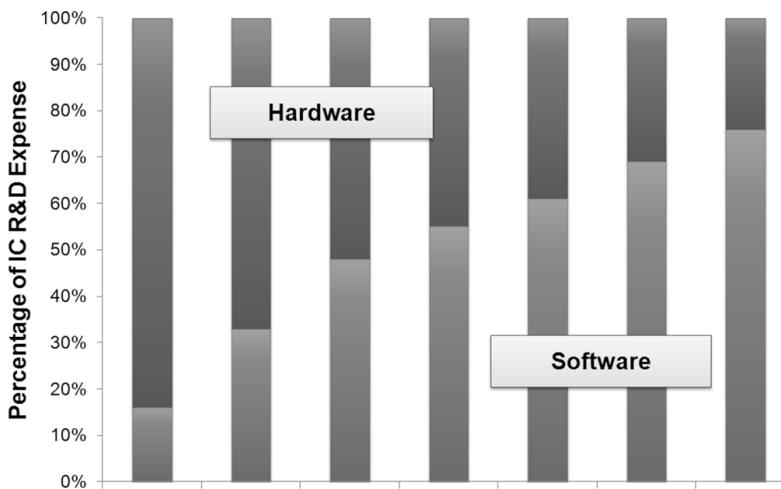


Figure 8 illustrates the projected software effort as percentage of R&D expense over technology nodes. At 65nm the expected R&D expense for software has surpassed that of hardware development.

Overall, software has become the critical path for chip development and its effort is surpassing that of hardware. In traditional serial design flows software development starts late, i.e., when hardware is well underway or even after final prototype chips are available. As a result software availability can hold up chip developments from reaching mainstream production.

From a prototyping perspective this represents yet another driver towards starting prototyping for software development as early as possible during a project. With software largely determining the functionality of a design, it is destined to also change verification flows. Software verification on prototypes will further gain in importance, as well as software becoming a driver for hardware verification too. As an alternative to classical verification using test benches coded in VHDL or SystemVerilog, directed tests using software have recently found more adoption.

Incidentally, this allows a new form of verification reuse across various phases of the development as described earlier.

To support this type of verification reuse, prototyping of the hardware as early as possible becomes mandatory. Given the seamless reuse of verification across various stages of the development, interfaces between different prototyping techniques have become more critical too. Virtual prototypes today can be connected to hardware-based prototypes to allow a mix of hardware- and software-based execution, offering a variety of advantages:

- **First**, avoiding having to re-model parts of the design which are already available in RTL reduces the development effort and enables hardware-assisted virtual platforms.
- **Second**, hardware prototypes can be brought up faster, because test benches – which traditionally can contain 50% of the overall defects – are already verified and stable as they have been applied to virtual prototypes before.
- **Third**, with a mix of hardware- and software-based techniques, trade-offs between accuracy, speed and time of availability of prototypes can be managed more flexibly.
- **Finally**, validation of the hardware/software prototype within the system context requires interfaces to the environment of the chip under development. Interfaces from hardware prototypes can execute close to, or even at, real-time. Interfaces using virtual prototypes can be made available even prior to actual hardware availability. For instance, USB 3.0 drivers were already developed on transaction-level models in virtual platforms, even before the actual cables were available.

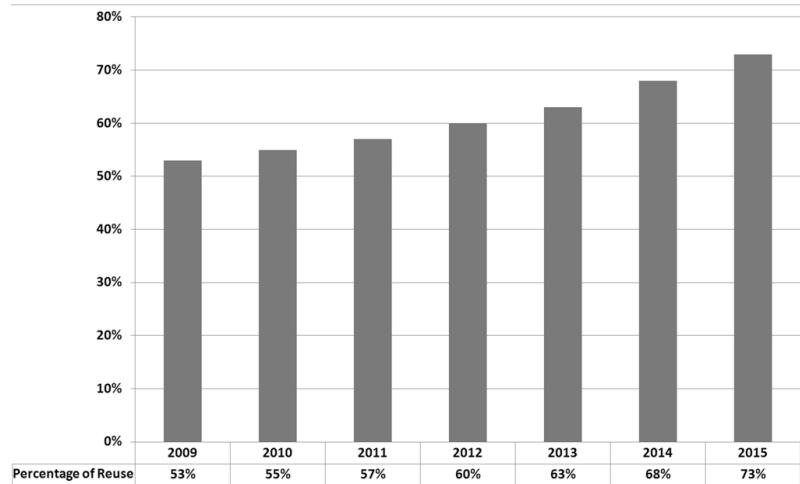
1.6.4. Intellectual property block reuse

Another important trend is the reuse of IP blocks. With growing chip complexity, IP reuse has become an essential way to maintain growth of design productivity. Figure 9 shows that the percentage of reuse continues to increase and although not shown on this graph, since 2007 the reuse of blocks has increased from 45% to 55% i.e., most blocks are now reused in other designs.

At the same time until the average number of IP blocks per chip has grown from 28 to 50, as shown in Figure 10. Both of these data points come from a study by Semico Research Corporation. Taking these IP trends into consideration, chip design itself is becoming a task of assembling existing blocks via interconnect fabrics. Chip differentiation can be achieved with custom blocks, custom co-processors and, of course, with software.

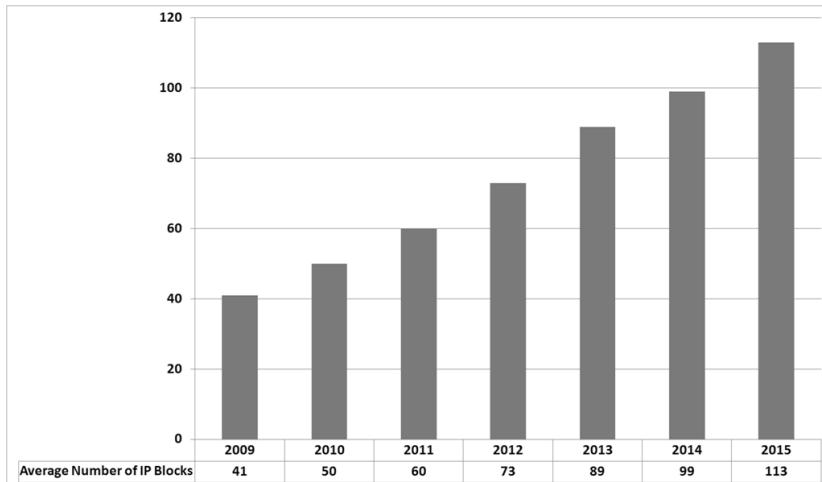
There are various effects of increased IP reuse on prototyping. First, pre-defined IP models are pre-mapped and pre-verified in FPGA prototypes to decrease bring-up time and reduce unnecessary duplication of work. Users of IP also increasingly

Figure 9: Percentage of reuse of IP blocks (Source: Semico Research Corp.)



request model libraries at different stages of the project and at different levels of abstraction as part of the IP delivery itself. This is already very common in the area of processors, for which IP providers like ARM®, MIPS®, ARC® and Tensilica® are asked by their users to provide processor models which can be used for early software development and verification.

Figure 10: IP instances in SoC designs (Source: Semico Research Corp.)



While in the past, development of those models was a challenge because they had to interface to various proprietary simulation environments, the development of such models has recently become commercially feasible.

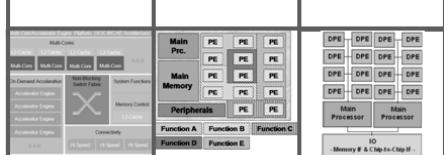
With the advent of standards like OSCI SystemC TLM-2.0, the models for processors, peripherals and interconnect have become interoperable across different SystemC-compliant simulation engines. Standardization implies the transition from an early adopter phase to mainstream, and as such availability of IP models has greatly improved.

1.6.5. Application specificity and mixed-signal design

The target application markets for which chips are developed have a profound impact on the chip development itself. Figure 11 summarizes some of the defining characteristics of different target applications, according to the International Technology Roadmap for Semiconductors (ITRS).

ITRS differentiates between four major categories of chip design – SoCs, microprocessor units (MPUs), mixed-signal design and embedded memory. Each of the categories has specific requirements. To keep the die area constant while increasing performance is important for MPUs. Decreasing supply voltages are a key issue for mixed signal. Within the SoC domain the ITRS separates networking applications from consumer portable and consumer stationary, with various sub-requirements as shown in Figure 11.

Figure 11: Application specific requirements (Source: ITRS)

SoC			MPU	Mixed Signal	Embedded Memory
Networking	Consumer Portable	Consumer Stationary			
<ul style="list-style-type: none"> Die area constant Number of cores increases by $1.4 \times / \text{year}$ Core frequency increases by $1.05 \times / \text{year}$ On-demand accelerator engine frequency increases by $1.05 \times / \text{year}$ Underlying fabrics scale consistently with the increase in number of cores. 	<ul style="list-style-type: none"> Rapid progress across technology generations. Rapid increase in processing capability, constraints on power. Processing power increases by $1000 \times$ in the next ten years, dynamic power consumption does not change significantly. Lifecycles short. Design effort cannot be increased. 	<ul style="list-style-type: none"> Processing performance most important, (2022 more than 120 TFlops) Functions mainly in software. High processing power required, Many Data Processing Engines (DPEs). Worse performance-to-power ratio, superior functional flexibility. Lifecycle relatively long, application area wide. 	<ul style="list-style-type: none"> Three types <ul style="list-style-type: none"> cost-performance (CP), reflecting "desktop." high-performance (HP), reflecting "server" power-connectivity-cost (PCC). Constant die area Multi-core organization Memory doubles every 18 months Layout density Maximum frequency 4.7GHz in 2007, grows at $1.25x$ per generation 	<ul style="list-style-type: none"> Interplay between cost and performance Decreasing supply voltage Increasing relative parametric variations Increasing numbers of analog transistors per chip Increasing processing speed (carrier or clock frequencies) Increasing crosstalk arising from SOC integration Shortage of design skills and productivity 	<ul style="list-style-type: none"> Code storage in reconfigurable applications (such as automotive) Data storage in smart or memory cards High memory content, high performance logic found in gaming or mass storage systems.
				Different Architecture Templates	

Overall, the end application has become more important for chip design requirements across the board and for SoCs specifically. As a result, prototyping in the different application domains requires application-specific analysis as well as application specific system interfaces, most of which have significant mixed signal content.

Besides other characteristics, the speed of external interfaces determines whether a prototype can be used directly or will need to be slowed down.

1.6.6. Multicore architectures and low power

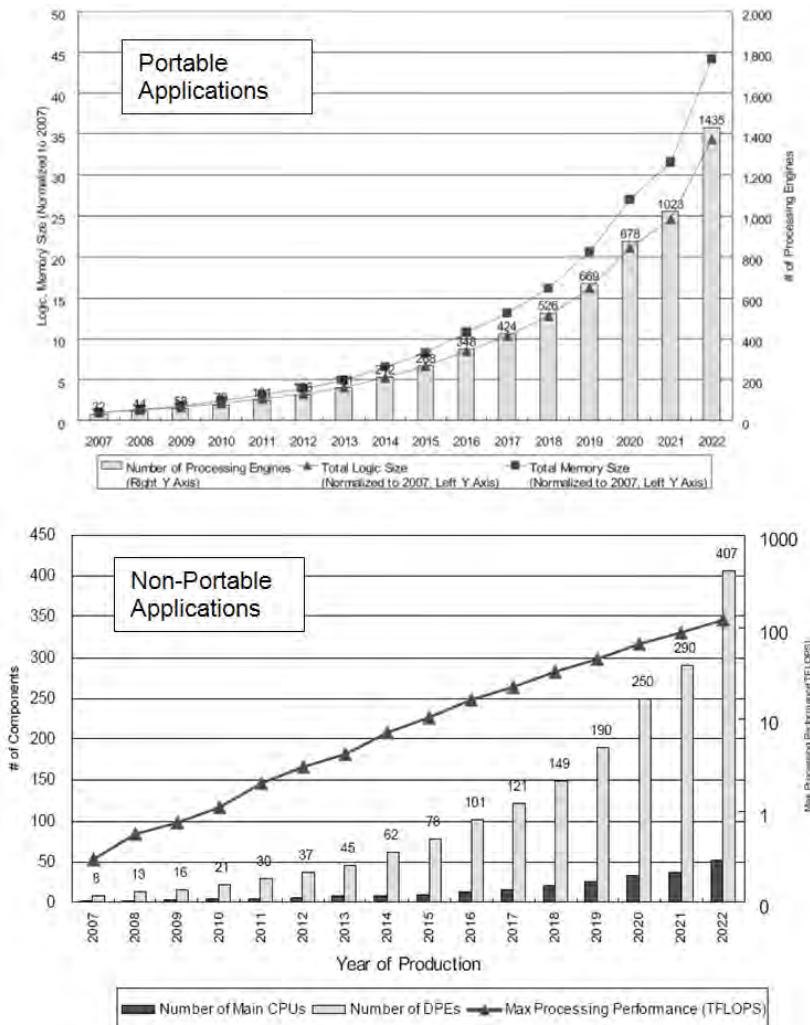
While for decades, the scaling of processors in speed has served the ever-increasing appetite of software applications for performance, the industry has run into limits around 4GHz for CPUs and 1GHz for embedded processors. The reason for that limitation lies in power consumption, which simply exceeds power envelopes when just scaled higher. This real and hard limitation has led to a trend to switch to multicore architectures. Simply put, more cores at lower frequency will result in less power consumption than simply scaling one core. The graphs in Figure 12 confirm this trend in CPU and data processing engines (DPE) usage for consumer applications, portable and non-portable.

We can see that the average number of DPEs, for example, has almost tripled from 2007 to 2011 and is expected to increase further. While this is a good solution on the hardware side, the challenge has now been transferred to the software side.

Traditionally, sequential software now needs to be distributed across multiple cores. For prototyping this means that debugging (the ability to look into the hardware/software execution) has become more important as well as the ability to start, pause, resume and stop hardware/software execution.

Today's virtual prototypes already offer intelligent techniques to un-intrusively debug design and they can be started and stopped at any given time. Demands on debug and control for hardware-based prototypes have also increased, but debug capabilities in FPGA-based prototypes still trail those of virtual prototypes.

Figure 12: CPU requirements for consumer applications (source: ITRS)



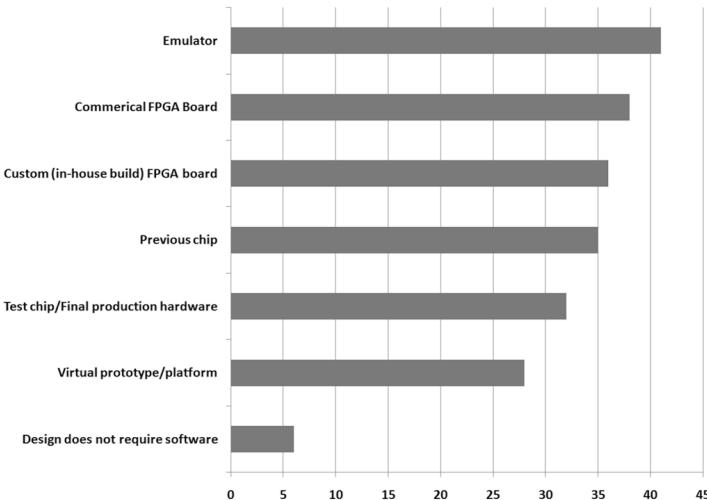
1.7. Summary

Prototyping, in all its forms, provides powerful methods for verifying the design of hardware and validating the software in models, which to a greater or lesser degree, mimic the target environment. FPGA-based prototyping is especially beneficial during the crucial later stages of the project when hardware and software are integrated for the first time. Users have several prototyping options and depending

on their main requirements can choose between various software- and hardware-based techniques to prototype their designs.

Owing to the fact that design verification and software development now dominate SoC development effort, the use of prototyping has never been more important in reducing project duration and design costs. The various IC trends mentioned above lead us also to only one conclusion: prototyping has become a necessary element of chip design and will become only more crucial in the future, as we shall see in the final chapter of this book.

Figure 13: Users recognizing a variety of prototyping options



In this chapter, we have introduced a lot of terminology and some quite different types of prototyping. How widespread are each of them in real life? To answer this we refer to Figure 14 which summarizes 116 responses to a usage survey made during the SoC Virtual Conference in August 2009. When asked the question “What method(s) do/will you use to develop hardware dependent software (e.g., drivers, firmware) for your design project?”, the results show that users do indeed recognize a variety of distinct prototyping solutions. The results indicate that all the prototyping techniques described earlier are in active use – a clear result of the different priorities as discussed above – favoring different prototyping options.

In the following chapter we will zoom in on the benefits of FPGA-based prototyping in particular to software teams and to the whole SoC project.

The authors gratefully acknowledge significant contribution to this chapter from
Frank Schirrmeister of Synopsys, Mountain View

As advocates for FPGA-based prototyping, we may be expected to be biased towards its benefits while being blind to its deficiencies, however, that is not our intent. The FPMM is intended to give a balanced view of the pros and cons of FPGA-based prototyping because, at the end of the day, we do not want people to embark on a long prototyping project if their aims would be better met by other methods, for example by using a SystemC-based virtual prototype.

This chapter will provide the aims and limitations of FPGA-based prototyping. After completing this chapter, readers should have a firm understanding of the applicability of FPGA-based prototyping to system-level verification and other aims. As we shall see in later chapters, by staying focused on the aim of the prototype project, we can simplify our decisions regarding platform, IP usage, design porting, debug etc. Therefore we can learn from the different aims that others have had in their projects by examining some examples from prototyping teams around the world.

2.1. FPGA-based prototyping for different aims

Prototyping is not a push-button process and requires a great deal of care and consideration at its different stages. As well as explaining the effort and expertise during the next few chapters, we should also give some incentive as to why we should (or maybe should not) perform prototyping during our SoC projects.

In conversation with prototypers over many years leading up to the creation of this book, one of the questions we liked to ask is “why do you do it?” There are many answers but we are able to group them into general reasons shown in Table 1. So for example, “real-world data effects” might describe a team that is prototyping in order to have an at-speed model of a system available to interconnect with other systems or peripherals, perhaps to test compliance with a particular new interface standard. Their broad reason to prototype is “interfacing with the real world” and prototyping does indeed offer the fastest and most accurate way to do that in advance of real silicon becoming available.

A structured understanding of these project aims and why we should prototype will help us to decide if FPGA-based prototyping is going to benefit our next project.

Let us, therefore, explore each of the aims in Table 1 and how FPGA-based prototyping can help. In many cases we shall also give examples from the real world and the authors wish to thank in advance those who have offered their own experiences as guides to others in this effort.

Table 1: General aims and reasons to use FPGA-based prototypes

Project Aim	Why Prototype?
Test real-time dataflow	High performance and accuracy
Early hardware-software integration	
Early software validation	
Test real-world data effects	Interfacing with the real world
Test real-time human interface	
Debug rare data dependencies	
Feasibility (proof of concept)	In-lab usage
Testing algorithms	
Public exhibitions	Out-of-lab demonstration
Encourage investors	
Extended RTL test and debug	Other aims

2.1.1. High performance and accuracy

Only FPGA-based prototyping provides both the speed and accuracy necessary to properly test many aspects of the design, as we shall describe.

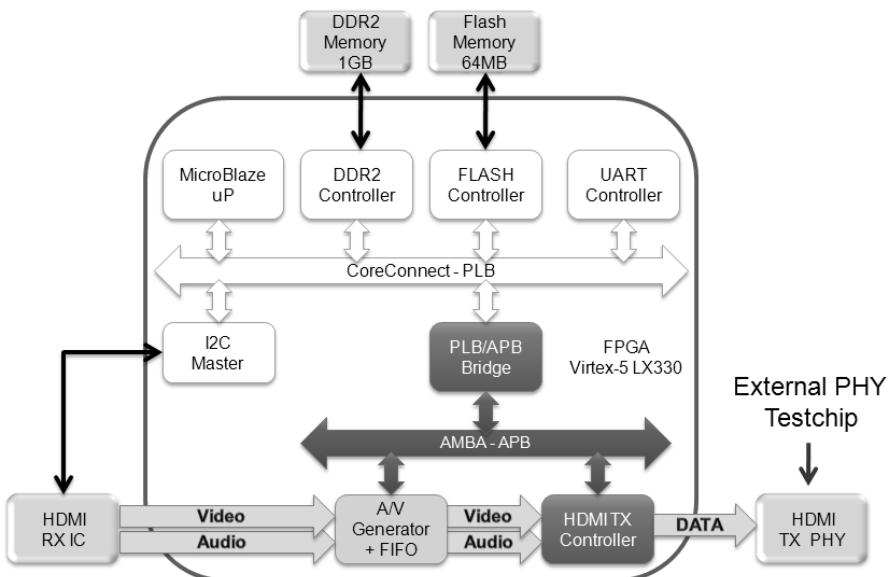
We put this reason at the top of the list because it is the most likely underlying reason of all for a team to be prototyping, despite the many different given deliverable aims of the project. For example, the team may aim to validate some of the SoC's embedded software and see how it runs at speed on real hardware, but the underlying reason to use a prototype is for both high performance and accuracy. We

could validate the software at even higher performance on a virtual system, but we lose the accuracy which comes from employing the real RTL.

2.1.2. Real-time dataflow

Part of the reason that verifying an SoC is hard is because its state depends upon many variables, including its previous state, the sequence of inputs and the wider system effects (and possible feedback) of the SoC outputs. Running the SoC design at real-time speed connected into the rest of the system allows us to see the immediate effect of real-time conditions, inputs and system feedback as they change.

Figure 14: Block diagram of HDMI prototype



A very good example of this is real-time dataflow in the HDMI™ prototype performed by the Synopsys IP group in Porto, Portugal. Here a high-definition (HD) media data stream was routed through a prototype of a processing core and out to an HD display, as show in the block diagram in Figure 14. We shall learn more about this design in chapter 10 when we consider IP usage in prototyping, but for the moment, notice that across the bottom of the diagram there is audio and HD video dataflow in real-time from the receiver (from an external source) through the prototype and out to a real-time HDMI PHY connection to an external monitor. By

using a pre-silicon prototype, we can immediately see and hear the effect of different HD data upon our design, and vice versa. Only FPGA-based prototyping allows this real-time dataflow, giving great benefits not only to such multimedia applications but to many other applications where real-time response to input dataflow is required.

2.1.3. Hardware-software integration

In the above example, readers may have noticed that there is a small MicroBlaze™ CPU in the prototype along with peripherals and memories, so all the familiar blocks of an SoC are present. In this design the software running in the CPU is used mostly to load and control the AV processing, however, in many SoC designs it is the software that requires most of the design effort.

Given that software has already come to dominate SoC development effort, it is increasingly common that the software effort is on the critical path of the project schedule. It is software development and validation that governs the actual completion date when the SoC can usefully reach volume production. In that case, what can system teams do to increase the productivity of software development and validation?

To answer this question, we need to see where software teams spend their time, which we will explore in the next sections.

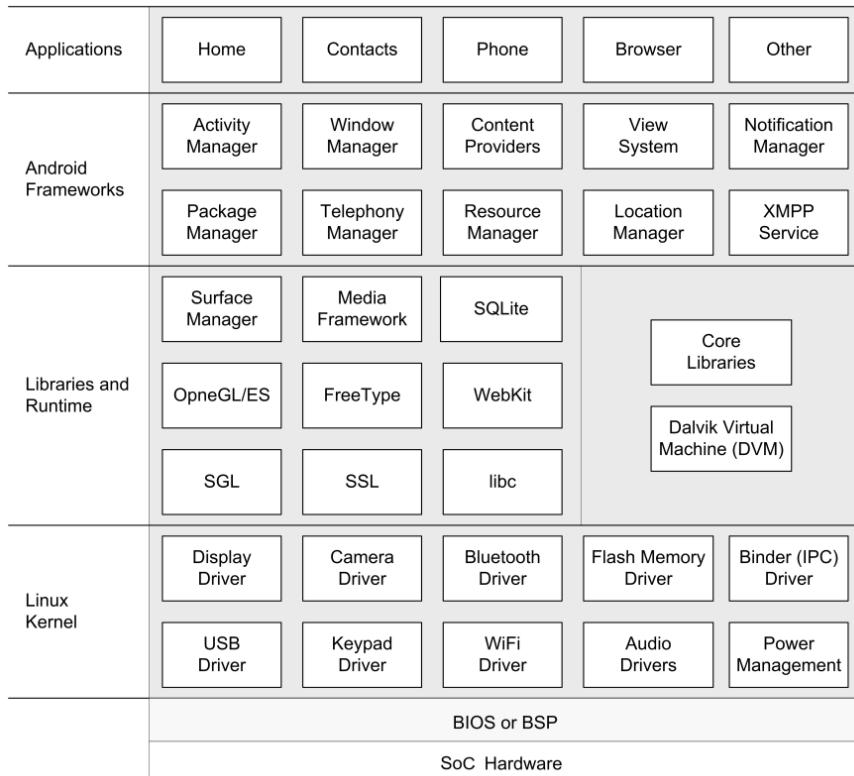
2.1.4. Modeling an SoC for software development

Software is complex and hard to make perfect. We are all familiar with the software upgrades, service packs and bug fixes in our normal day-to-day use of computers. However, in the case of software embedded in an SoC, this perpetual fine tuning of software is less easily achieved. On the plus side, the system with which the embedded software interacts, its intended use modes and the environmental situation are all usually easier to determine than for more general-purpose computer software. Furthermore, embedded software for simpler systems can be kept simple itself and so easier to fully validate. For example, an SoC controlling a vehicle subsystem or an electronic toy can be fully tested more easily than a smartphone which is running many apps and processes on a real-time operating system (RTOS).

If we look more closely at the software running in such a smartphone, for example the Android™ software shown in Figure 15, then we see a multi-layered arrangement, called a software stack. This diagram is based on an original by software designer Frank Abelson in his book “*Unlocking Android*.”

Taking a look at the stack, we should realize that the lowest levels i.e., those closest to the hardware, are dominated by the need to map the software onto the SoC hardware. This requires absolute knowledge of the hardware to an address and clock-cycle level of accuracy. Designers of the lowest level of a software stack, often calling themselves platform engineers, have the task of describing the hardware in terms that the higher levels of the stack can recognize and reuse. This description is called a BSP (board support package) by some RTOS vendors and is also analogous to the BIOS (basic input/output system) layer in our day-to-day PCs.

Figure 15: The Android™ stack (based on source: “*Understanding Android*”)



The next layer up from the bottom of the stack contains the kernel of the RTOS and the necessary drivers to interface the described hardware with the higher level software. In these lowest levels of the stack, platform engineers and driver developers will need to validate their code on either the real SoC or a fully accurate model of the SoC. Software developers at this level need complete visibility of the behavior of their software at every clock cycle.

At the other extreme for software developers, at the top layer of the stack, we find the user space which may be running multiple applications concurrently. In the smartphone example these could be a contact manager, a video display, an internet browser and of course, the phone sub-system that actually makes calls. Each of these applications does not have direct access to SoC hardware and is actually somewhat divorced from any consideration of the hardware. The applications rely on software running on lower levels of the stack to communicate with the SoC hardware and the rest of the world on its behalf.

We can generalize that, at each layer of the stack, a software developer only needs a model with enough accuracy to fool his own code into thinking it is running in the target SoC. More accuracy than necessary will only result in the model running more slowly on the simulator. In effect, SoC modeling at any level requires us to represent the hardware and the stack up to the layer just below the current level to be validated and optimally, we should work with just enough accuracy to allow maximum performance.

For example, application developers at the top of the stack can test their code on the real SoC or on a model. In this case the model need only be accurate enough to fool the application into thinking that it is running on the real SoC, i.e., it does not need cycle accuracy or fine-grain visibility of the hardware. However, speed is important because multiple applications will be running concurrently and interfacing with real-world data in many cases.

This approach of the model having “just enough accuracy” for the software layer gives rise to a number of different modeling environments being used by different software developers at different times during an SoC project. It is possible to use transaction-level simulations, modeled in languages such as SystemC™, to create a simulator model which runs with low accuracy but at high enough speed to run many applications together. If handling of real-time, real-world data is not important then we might be better considering such a virtual prototyping approach.

However, FPGA-based prototyping becomes most useful when the whole software stack must run together or when real-world data must be processed.

2.1.5. Example prototype usage for software validation

Only FPGA-based prototyping breaks the inverse relationship between accuracy and performance inherent in modeling methodologies. By using FPGAs we can achieve speeds up to real-time and yet still be modeling at the full RTL cycle accuracy. This enables the same prototype to be used not only for the accurate models required by low-level software validation but also for the high-speed models needed by the high-level application developers. Indeed, the whole SoC software stack can be modeled on a single FPGA-based prototype. A very good example of this software validation using FPGAs is seen in a project performed by Scott

Constable and his team at Freescale® Semiconductor's Cellular Products Group in Austin, Texas.

Freescale was very interested in accelerating SoC development because short product life cycles of the cellular market demand that products get to market quickly not only to beat the competition, but also to avoid quickly becoming obsolete. Analyzing the biggest time sinks in its flow, Freescale decided that the greatest benefit would be achieved by accelerating their cellular 3G protocol testing. If this testing could be performed pre-silicon, then Freescale would save considerable months in a project schedule. When compared to a product lifetime that is only one or two years this is very significant indeed.

Protocol testing is a complex process that even at high real time speeds require a day to complete. Using RTL simulation would take years and running on a faster emulator would still have taken weeks, neither of which was a practical solution. FPGAs were chosen because that was the only way to achieve the necessary clock speed to complete the testing in a timely manner.

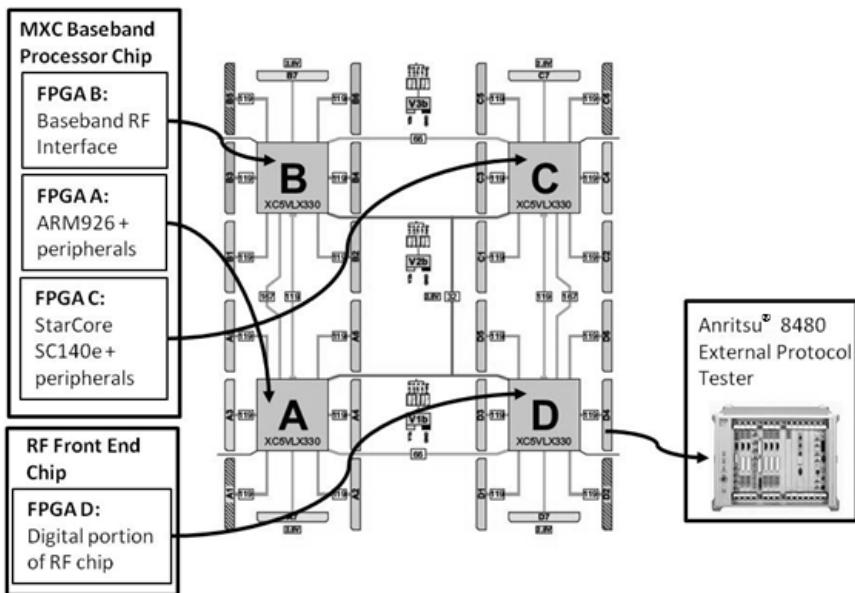
Protocol testing requires the development of various software aspects of the product including hardware drivers, operating system, and protocol stack code. While the main goal was protocol testing, as mentioned, by using FPGAs all of these software developments would be accomplished pre-silicon and greatly accelerate various end product schedules.

Freescale prototyped a multichip system that included a dual core MXC2 baseband processor plus the digital portion of an RF transceiver chip. The baseband processor included a Freescale StarCore® DSP core for modem processing and an ARM926™ core for user application processing, plus more than 60 peripherals.

A Synopsys HAPS®-54 prototype board was used to implement the prototype, as shown in Figure 16. The baseband processor was more than five million ASIC gates and Scott's team used Synopsys Certify® tools to partition this into three of the Xilinx® Virtex®-5 FPGAs on the board while the digital RF design was placed in the fourth FPGA. Freescale decided not to prototype the analog section but instead delivered cellular network data in digital form directly from an Anritsu™ protocol test box.

Older cores use some design techniques that are very effective in an ASIC, but they are not very FPGA friendly. In addition, some of the RTL was generated automatically from system-level design code which can also be fairly unfriendly to FPGAs owing to over-complicated clock networks. Therefore, some modifications had to be made to the RTL to make it more FPGA compatible but the rewards were significant.

Figure 16: The Freescale® SoC design partitioned into HAPS® -54 board



Besides accelerating protocol testing, by the time Freescale engineers received first silicon they were able to:

- Release debugger software with no major revisions after silicon.
- Complete driver software.
- Boot up the SoC to the OS software prompt.
- Achieve modem camp and registration.

The Freescale team was able to reach the milestone of making a cellular phone call through the system only one month after receipt of first silicon, accelerating the product schedule by over six months.

To answer our question about what FPGA-based prototyping can do for us, let's hear in Scott Constable's own words:

"In addition to our stated goals of protocol testing, our FPGA system prototype delivered project schedule acceleration in many other areas, proving its worth many times over. And perhaps most important was the immeasurable human benefit of getting engineers involved earlier in the project schedule, and having all teams from design to software to validation to applications very familiar with the product six months before

silicon even arrived. The impact of this accelerated product expertise is hard to measure on a Gantt chart, but may be the most beneficial.

“In light of these accomplishments using an FPGA prototype solution to accelerate ASIC schedules is a “no-brainer.” We have since spread this methodology into the Freescale Network and Microcontroller Groups and also use prototypes for new IP validation, driver development, debugger development, and customer demos.”

This example shows how FPGA-based prototyping can be a valuable addition to the software team’s toolbox and brings significant return on investment in terms of product quality and project timescales.

2.2. Interfacing benefit: test real-world data effects

It is hard to imagine an SoC design that does not comply with the basic structure of having input data upon which some processing is performed in order to produce output data. Indeed, if we push into the SoC design we will find numerous sub-blocks which follow the same structure, and so on down to the individual gate level.

Verifying the correct processing at each of these levels requires us to provide a complete set of input data and to observe that the correct output data are created as a result of the processing. For an individual gate this is trivial, and for small RTL blocks it is still possible. However, as the complexity of a system grows it soon becomes statistically impossible to ensure completeness of the input data and initial conditions, especially when there is software running on more than one processor.

There has been huge research and investment in order to increase efficiency and coverage of traditional verification methods and to overcome the challenge of this complexity. At the complete SoC level, we need to use a variety of different verification methods in order to cover all the likely combinations of inputs and to guard against unlikely combinations.

This last point is important because unpredictable input data can upset all but the most carefully designed critical SoC-based systems. The very many possible previous states of the SoC coupled with new input data, or with input data of an unusual combination or sequence, can put an SoC into a non-verified state. Of course that may not be a problem and the SoC recovers without any other part of the system, or indeed the user, becoming aware.

However, unverified states are to be avoided in final silicon and so we need ways to test the design as thoroughly as possible. Verification engineers use powerful methods such as constrained-random stimulus and advanced test harnesses to perform a very wide variety of tests during functional simulations of the design, aiming to reach an acceptable coverage. However, completeness is still governed by the direction and constraints given by the verification engineers and the time

available to run the simulations themselves. As a result, constrained-random verification is never fully exhaustive but it will greatly increase confidence that we have tested all combinations of inputs, both likely and unlikely.

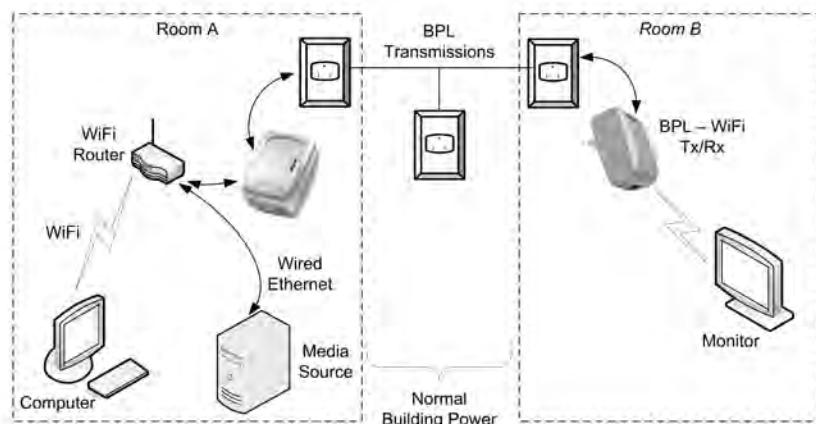
In order to guard against corner case combinations we can complement our verification results with observations of the design running on an FPGA-based prototype running in the real world. By placing the SoC design into a prototype, we can run at a speed and accuracy point which compares very well with the final silicon, allowing “soak” testing within the final ambient data, much as would be done with the final silicon.

One example of this immersion of the SoC design into a real-world scenario is the use made of FPGA-based prototyping at DS2 in Valencia, Spain.

2.2.1. Example: prototype immersion in real-world data

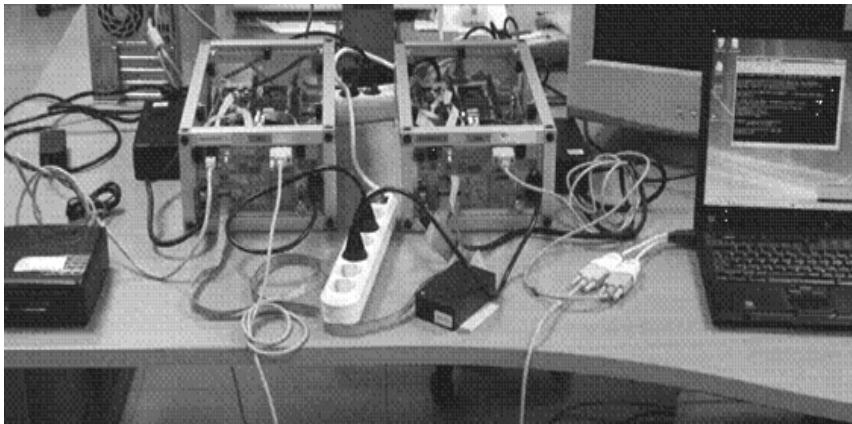
Broadband-Over-Powerline (BPL) technology uses normally undetectable signals to transmit and receive information over electrical mains powerlines. A typical use of BPL is to distribute HD video around a home from a receiver to any display via the mains wiring, as shown in Figure 17..

Figure 17: BPL technology used in WiFi Range Extender



At the heart of the DS2's BPL designs lay sophisticated algorithms in hardware and embedded software which encode and retrieve the high-speed transmitted signal into and out of the powerlines. These powerlines can be very noisy electrical environments so a crucial part of the development is to verify these algorithms in a wide variety of real-world conditions, as shown in Figure 18.

Figure 18: DS2 making in situ tests on real-world data (source: DS2)



Javier Jimenez, ASIC Design Manager at DS2 explains what FPGA-based prototyping did for them . . .

“It is necessary to use a robust verification technology in order to develop reliable and high-speed communications. It requires very many trials using different channel and noise models and only FPGA-based prototypes allow us to fully test the algorithms and to run the design’s embedded software on the prototype. In addition, we can take the prototypes out of the lab for extensive field testing. We are able to place multiple prototypes in real home and workplace situations, some of them harsh electrical environments indeed. We cannot consider emulator systems for this purpose because they are simply too expensive and are not portable.”

This usage of FPGA-based prototyping outside of the lab is instructive because we see that making the platform reliable and portable is crucial to success. We explore this further in chapters 5 and 12.

2.3. Benefits for feasibility lab experiments

At the beginning of a project, fundamental decisions are made about chip topology, performance, power consumption and on-chip communication structures. Some of these are best performed using algorithmic or system-level modeling tools but some extra experiments could also be performed using FPGAs. Is this really FPGA-based prototyping? We are using FPGAs to prototype an idea but it is different to using algorithmic or mathematical tools because we need some RTL, perhaps generated by those high-level tools. Once in FPGA, however, early information can be

gathered to help drive the optimization of the algorithm and the eventual SoC architecture. The extra benefit that FPGA-based prototypes bring to this stage of a project is that more accurate models can be used which can run fast enough to interact with real-time inputs.

Experimental prototypes of this kind are not the main subject of this book but are worth mentioning as they are another way to use FPGA-based prototyping hardware and tools in between full SoC projects, hence deriving further return on our investment.

2.4. Prototype usage out of the lab

One truly unique aspect of FPGA-based prototyping for validating SoC design is its ability to work standalone. This is because the FPGAs can be configured, perhaps from a flash EEPROM card or other self-contained medium, without supervision from a host PC. The prototype can therefore run standalone and be used for testing the SoC design in situations quite different to those provided by other modeling techniques, such as emulation, which rely on host intervention.

In extreme cases, the prototype might be taken completely out of the lab and into real-life environments in the field. A good example of this might be the ability to mount the prototype in a moving vehicle and explore the dependency of a design to variations in external noise, motion, antenna field strength and so forth. For example, the authors are aware of mobile phone baseband prototypes which have been placed in vehicles and used to make on-the-move phone calls through a public GSM network.

Chip architects and other product specialists need to interact with early adopter customers and demonstrate key features of their algorithms. FPGA-based prototyping can be a crucial benefit at this very early stage of a project but the approach is slightly different to the mainstream SoC prototyping.

Another very popular use of FPGA-based prototypes out of the lab is for pre-production demonstration of new product capabilities at trade shows. We will explore the specific needs for using a prototype outside of the lab in Chapter 12 but for now let's consider a use of FPGA-based prototyping by the Research and Development division of The BBC in England (yes, *that* BBC) which illustrates both out-of-lab usage and use at a trade-show.

2.4.1. Example: A prototype in the real world

The powerful ability of FPGAs to operate standalone is demonstrated by a BBC Research & Development project to launch DVB-T2 in the United Kingdom. DVB-

T2 is a new, state-of-the-art open standard, which allows HD television to be broadcast from terrestrial transmitters.

The reason for using FPGA-based prototyping was that, like most international standards, the DVB-T2 technical specification took several years to complete, in fact 30,000 engineer-hours by researchers and technologists from all over the world. Only FPGAs gave the flexibility required in case of changes along the way. The specification was frozen in March 2008 and published three months later as a DVB Blue Book on 26 June 2008.

Because the BBC was using FPGA-based prototyping, in parallel with the specification work, a BBC implementation team, led by Justin Mitchell from BBC Research & Development, was able to develop a hardware-based modulator and demodulator for DVB-T2.

The modulator, shown in Figure 19, is based on a Synopsys HAPS®-51 card with a Virtex-5 FPGA from Xilinx. The HAPS-51 card was connected to a daughter card that was designed by BBC Research & Development. This daughter card provided an ASI interface to accept the incoming transport stream. The incoming transport stream was then passed to the FPGA for encoding according to the DVB-T2 standard and passed back to the daughter card for direct up-conversion to UHF.

Figure 19: DVB-T2 prototype at BBC Research and Development (Source: BBC)



The modulator was used for the world's first DVB-T2 transmissions from a live TV transmitter, which were able to start the same day that the specification was published.

The demodulator, also using HAPS as a base for another FPGA-based prototype, completed the working end-to-end chain and this was demonstrated at the IBC exhibition in Amsterdam in September 2008, all within three months of the specification being agreed. This was a remarkable achievement and helped to build confidence that the system was ready to launch in 2009.

BBC Research & Development also contributed to other essential strands of the DVB-T2 project including a very successful "PlugFest" in Turin in March 2009, at which five different modulators and six different demodulators were shown to work together in a variety of modes. The robust and portable construction of the BBC's prototype made it ideal for this kind of PlugFest event.

Justin explains what FPGA-based prototyping did for them as follows:

"One of the biggest advantages of the FPGA was the ability to track late changes to the specification in the run up to the transmission launch date. It was important to be able to make quick changes to the modulator as changes were made to the specification. It is difficult to think of another technology that would have enabled such rapid development of the modulator and demodulator and the portability to allow the modulator and demodulator to be used standalone in both a live transmitter and at a public exhibition."

2.5. What can't FPGA-based prototyping do for us?

We started this chapter with the aim of giving a balanced view of the benefits and limitations of FPGA-based prototyping, so it is only right that we should highlight here some weaknesses to balance against the previously stated strengths.

2.5.1. An FPGA-based prototype is not a simulator

First and foremost, an FPGA prototype is not an RTL simulator. If our aim is to write some RTL and then implement it in an FPGA as soon as possible in order to see if it works, then we should think again about what is being bypassed. A simulator has two basic components; think of them as the engine and the dashboard. The engine has the job of stimulating the model and recording the results. The dashboard allows us to examine those results. We might run the simulator in small increments and make adjustments via our dashboard, we might use some very sophisticated stimulus – but that's pretty much what a simulator does. Can an FPGA-based prototype do the same thing? The answer is no.

It is true that the FPGA is a much faster engine for running the RTL "model," but when we add in the effort to setup that model (i.e., the main content of this book) then the speed benefit is soon swamped. On top of that, the dashboard part of the simulator offers complete control of the stimulus and visibility of the results. We

shall consider ways to instrument an FPGA in order to gain some visibility into the design's functionality, but even the most instrumented design offers only a fraction of the information that is readily available in an RTL simulator dashboard. The simulator is therefore a much better environment for repetitively writing and evaluating RTL code and so we should always wait until the simulation is mostly finished and the RTL is fairly mature before passing it over to the FPGA-based prototyping team. We consider this hand-over point in more detail in chapter 4.

2.5.2. An FPGA-based prototype is not ESL

As we described in our introduction, electronic system-level (ESL) or algorithmic tools such as Synopsys's Innovator or Symphony, allow designs to be entered in SystemC or to be built from a library of pre-defined models. We then simulate these designs in the same tools and explore their system-level behavior including running software and making hardware-software trade-offs at an early stage of the project.

To use FPGA-based prototyping we need RTL, therefore it is not the best place to explore algorithms or architectures, which are not often expressed in RTL. The strength of FPGA-based prototyping for software is when the RTL is mature enough to allow the hardware platform to be built and then software can run in a more accurate and real-world environment. There are those who have blue-sky ideas and write a small amount of RTL for running in an FPGA for a feasibility study, as mentioned previously in section 2.3. This is a minor but important use of FPGA-based prototyping, but is not to be confused with running a system-level or algorithmic exploration of a whole SoC.

2.5.3. Continuity is the key

Good engineers always choose the right tool for the job, but there should always be a way to hand over work-in-progress for others to continue. We should be able to pass designs from ESL simulations into FPGA-based prototypes with as little work as possible. Some ESL tools also have an implementation path to silicon using high-level synthesis (HLS), which generates RTL for inclusion in the overall SoC project. An FPGA-based prototype can take that RTL and run it on a board with cycle accuracy but once again, we should wait until the RTL is relatively stable, which will be after completion of the project's hardware-software partitioning and architectural exploration phase.

In chapter 13, we shall explore ways that FPGA-based prototypes can be linked into ESL and RTL simulations. The prototype can supplement those simulations but cannot really replace them and so we will focus in this book on what FPGA-based prototyping can do really well.

2.6. Summary: So why use FPGA-based prototyping?

Today's SoCs are a combination of the work of many different experts from algorithm researchers, to hardware designers, to software engineers, to chip layout teams and each has their own needs as the project progresses. The success of an SoC project depends to a large degree on the hardware verification, hardware-software co-verification and software validation methodologies used by each of the above experts. FPGA-based prototyping brings different benefits to each of these experts:

For the hardware team, the speed of verification tools plays a major role in verification throughput. In most SoC developments it is necessary to run through many simulations and repeated regression tests as the project matures. Emulators and simulators are the most common platforms used for that type of RTL verification. However, some interactions within the RTL or between the RTL and external stimuli cannot be recreated in a simulation or emulation owing to long runtime, even when TLM-based simulation and modeling is used.. FPGA-based prototyping is therefore used by some teams to provide a higher performance platform for such hardware testing. For example, we can run a whole OS boot in relatively real-time, saving days of simulation time to achieve the same thing.

For the software team, FPGA-based prototyping provides a unique pre-silicon model of the target silicon, which is fast and accurate enough to enable debug of the software in near-final conditions.

For the whole team, a critical stage of the SoC project is when the software and hardware are introduced to each other for the first time. The hardware will be exercised by the final software in ways that were not always envisaged or predicted by the hardware verification plan in isolation, exposing new hardware issues as a result. This is particularly prevalent in multicore systems or those running concurrent real-time applications. If this hardware-software introduction were to happen only after first silicon fabrication then discovering new bugs at that time is not ideal, to put it mildly.

An FPGA-based prototype allows the software to be introduced to a cycle-accurate and fast model of the hardware as early as possible. SoC teams often tell us that the greatest benefit of FPGA-based prototyping is that when first silicon is available, the system and software are up and running in a day.

The authors gratefully acknowledge significant contribution to this chapter from

Scott Constable of Freescale Semiconductor, Austin, Texas

Javier Jimenez of DS2, Valencia, Spain

Justin Mitchell of BBC Research & Development, London, England

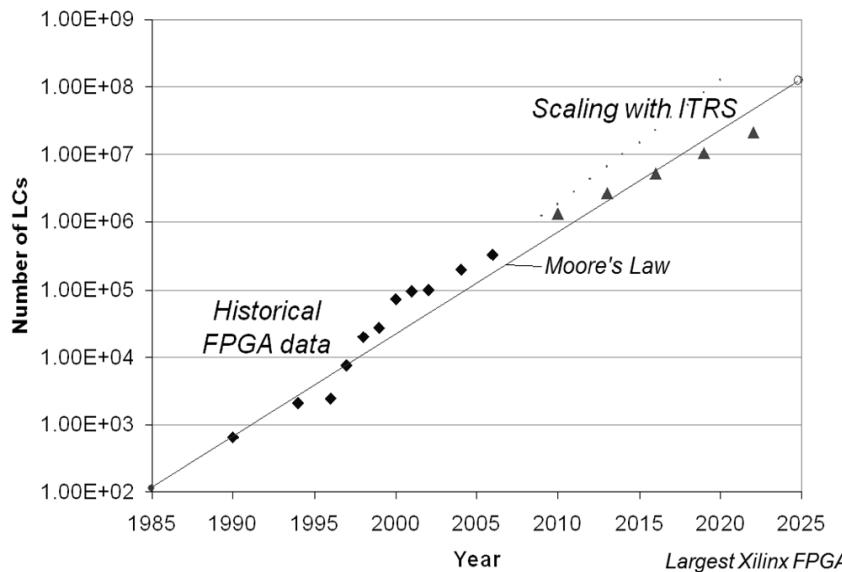
This chapter's focus is on the available technology, both hardware and software, for FPGA-based prototyping. It describes the main features of FPGAs as core technology and associated synthesis software technology as they pertain to FPGA-based prototyping. Following chapters describe how to use these technologies in greater detail. Firstly, describing current FPGA technology in general but focusing on the Xilinx® Virtex®-6 family. We will highlight the utility of each feature for FPGA-based prototyping, which depends not only on its functionality but also on its support in the relevant EDA tool.

3.1. FPGA device technology today

FPGAs devices are at the heart of the FPGA-based prototyping physical implementation. It is where the SoC design is going to be mapped and realized, so we really should take a close look at them and at the tools we use to work with them.

FPGAs have evolved over the years from modest (by today's standard) programmable logic devices to very large logic arrays with versatile architectural features, running at impressive clock rates. A glimpse at Figure 20 shows in particular, the inexorable progress of FPGA capacity as architectures have improved and silicon technology has evolved in accordance with Moore's Law. Indeed, those fabrication lines associated with producing FPGA devices for the main FPGA vendors, all of whom are fabless, have benefited from the experience of producing very large FPGA die and have been able to tune their processes accordingly. It should therefore be no surprise that FPGA progress has been in lockstep with the progress of silicon technology as a whole and we should expect it to at least continue to do so. In fact, at the time of writing, some exciting new developments are taking place with the use of 3D IC technology to allow some FPGAs to leap beyond Moore's Law.

Figure 20: The evolution of FPGA technology



Copyright © 2011 Xilinx, Inc.

Investing in an FPGA-based prototyping approach should not be seen as involving any risk because of a scarcity of FPGA technology itself.

Let's take a close look at a leading edge technology today; the Virtex®-6 family from Xilinx.

3.1.1. The Virtex®-6 family: an example of latest FPGAs

As our example in this chapter, we shall focus on Xilinx® FPGAs, since as of the writing of this chapter they are the most popular choice for FPGA-based prototyping across a wide section of the industry. The Xilinx® Virtex-6 family is currently the latest FPGA family from Xilinx, evolving from the Xilinx® Virtex®-5 architecture, but with enhanced features, greater capacity, improved performance and better power consumption.

As seen in Table 2, fabrication in a smaller CMOS process geometry enables more than doubling of logic capacity between the largest Virtex-5 and Virtex-6 devices but in addition, the ratio of FF (flip-flop) to logic resources has more than doubled, enabling better support for pipelined designs.

Table 2: Comparing largest Xilinx® Virtex®-5 and Virtex®-6 devices

Feature	Virtex®-5	Virtex®-6
Logic Cells	360,000	760,000
FFs	207,000	948,000
BlockRAM	18 MB	38 MB

Complete details of the devices and architectures is available by using some of the resources in the bibliography and appendices of this book, but let us spend some time now understanding each part of the FPGA technology, starting with the basic logic blocks and how helpful each may be towards our task of FPGA-based prototyping.

3.1.2. FPGA logic blocks

Sequential and combinatorial logic is implemented in logic blocks called slices. Slices contain look-up tables (LUTs), storage elements, and additional cascading logic.

Far more detail about FPGA technology is available in the references, but since the LUT is the fundamental building block in large FPGAs, it is worth a short examination here.

A typical 4-input LUT is at its heart a 16x1 RAM. Any particular bit of the RAM will be routed to the LUT's output depending upon the 4-bit address. Now consider filling the 16 bits of RAM with various 1s and 0s so that when the address changes, so will the LUT output. We have created a logic function of four inputs and one output. All that remains is to so order the 16 bits of RAM to mimic a useful logic function. In fact, we can consider the 16 bits of RAM as a Karnaugh map and in the very early days, that was actually an option for programming them.

Thus logic functions, such as parity, XOR, AND, OR, and so forth, may be efficiently packed into the smallest number of LUTs to perform the desired function. Arithmetic functions may also be placed in LUTs, and there is also hardwired carry look-ahead logic in the device so that performance may be improved over the use of LUTs alone.

Nowadays, we have up to 6-input LUTs (hence 64 bits of RAM) and it is all “programmed” via synthesis, which creates the LUT contents for us from a high-level description as required. Then the LUT RAM is loaded upon device configuration to create a complex 6-input function from just 64 bits of RAM. LUTs

are embedded into other structures which include FFs, carry chains, arithmetic, memories and other sophisticated structures.

In a Xilinx® Virtex-6 device, LUTs are used to implement function generators of six independent inputs. Each six-input LUT has two outputs. These function generators can implement any arbitrarily defined boolean function of up to six inputs for the first output, and up to five inputs for the second output of the same LUT.

Two slices are combined into a configurable logic block (CLB). CLBs are arranged in the FPGA in an array, and are connected to each other and to other types of block via interconnect resources.

In Virtex-6 devices there are two types of slices:

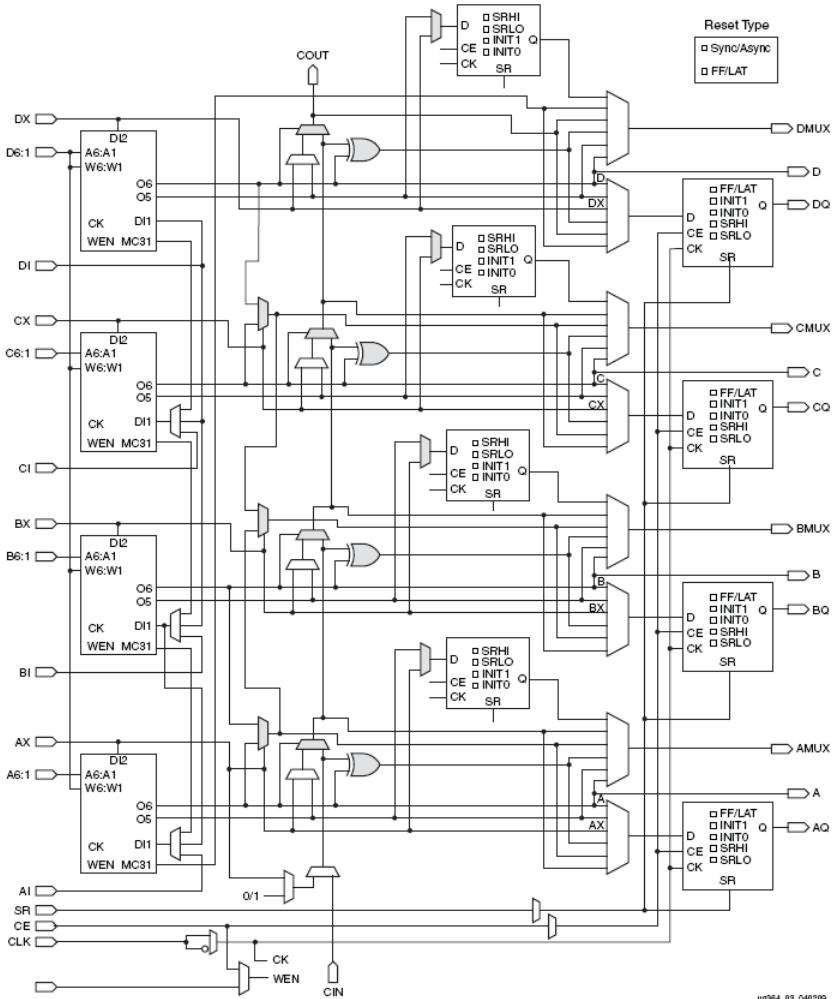
- SLICEM - a slice in which its LUTs can be used to implement either combinatorial functions, a small RAM block or a shift register.
- SLICEL - a slice in which its LUTs can be used to implement combinatorial logic only.

Figure 21 shows the SLICEM block diagram in which we can see that each slice contains four 6-input LUTs (on the left), eight storage elements (four FFs and four FF/Latches), and cascading logic. The various paths in the CLB can be programmed to connect or by-pass various combinations of the LUTs and FFs. Closer inspection also shows additional logic gates for particular carry and cascading functions which link resources within and outside the CLBs.

SLICEL is similar with the exception that the LUTs have only six input and two output signals. These resources can be configured for use as memory, most commonly RAM, and this is described briefly in section 3.1.3 below.

If the tools can make optimal use of the CLB in order to implement the design then the prototype will probably use less FPGA resources and run faster. This means that the tools must understand all the ways that the slice can be configured, and also what restrictions there may be on the use of the slice. For example, if the four FF/LAT storage elements are configured as latches, then the other four FFs cannot be used, hence designs which do not use latches are preferred. Also, control signals to the registers are shared so packing of design registers into slices becomes a complex task for the place & route tools if there are many different unrelated control signals in the design.

Figure 21: The Xilinx® Virtex®-6 SLICEM block diagram



Copyright © 2011 Xilinx, Inc.

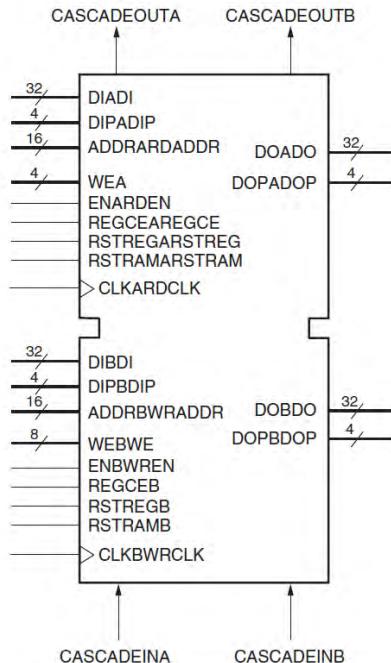
As prototypers, we typically do not concern ourselves with the final usage of the various features of the slice because the implementation tools should automatically use an appropriate configuration in order to meet timing constraints. Having expertise to that level of detail may sometimes be necessary, however, so we recommend the references at the end of this book for further information.

Prototyping utility: very high, the essential building block. Synthesis will make good use of all the above features either automatically or under the direction of optional attributes and directives in the RTL and/or constraint files.

3.1.3. FPGA memory: LUT memory and block memory

SoC designs include multiple memories of various types, e.g., RAM, ROM, content-addressable. In the vast majority of cases, these will instantiated memories either from a cell library or from a memory generator utility. It is important that the FPGA can represent these memories as efficiently as possible. A selection of memory types are available in most high-end FPGAs, from small register files and shift registers, up to large scale RAMs. As we saw in section 3.1.2, the LUT in a Xilinx® Virtex-6 SLICEM logic block may be employed as a small local memory, for example, as a 32-bit bit-wide RAM. This allows significant freedom to implement the function of small memories found in many places in SoC designs.

Figure 22: Xilinx® Virtex® - 6 BlockRAM



Copyright © 2011 Xilinx, Inc.

For the largest SoC memories, external memory resources are required. The FPGA's own block memory resources will be very useful for smaller memories, and from a prototyping perspective, they are the second most critical resource in an FPGA. In the case of the Virtex-6 family, this memory resource is called BlockRAM and there are between 156 and 1064 BlockRAMs distributed throughout a Virtex-6 FPGA device.

A diagram of the Xilinx® Virtex-6 BlockRAM is shown in Figure 22

BlockRAMs have the following main features:

- **Configurability:** each block is a dedicated, dual-ported synchronous 36 Kbits RAM block that can be configured as $32K \times 1$, $16K \times 2$, $8K \times 4$, $4K \times 9$ (or 8), $2K \times 18$ (or 16), $1K \times 36$ (or 32), or 512×72 (or 64). Each port can be configured independently of the other.
- **Synchronous operation:** BlockRAMs can implement any single or dual ported synchronous memory. When configured as dual-ported RAM, each port can operate at a different clock rate.
- **FIFO logic:** dedicated – yet configurable – FIFO logic can be used in combination with BlockRAMs to implement address points and handshaking flags. FIFO logic's depth and width can be configurable but both write and read sides must be the same width.
- **ECC:** when configured to 64-bit wide, each BlockRAM can store and utilize eight additional Hamming-code bits and perform single-bit error correction and double-bit error detection (ECC) during the read process. The ECC logic can also be used when writing to, or reading from external 64/72-bit wide memories.

BlockRAMs in the FPGA can be combined to model either deeper or wider memory SoC memories. This is commonly performed by the synthesis tools, which automatically partition larger memories into the multiple BlockRAMs. Some manipulation of the design from the SoC instantiation into the final FPGA BlockRAMs will be required and this is covered in detail in chapter 7.

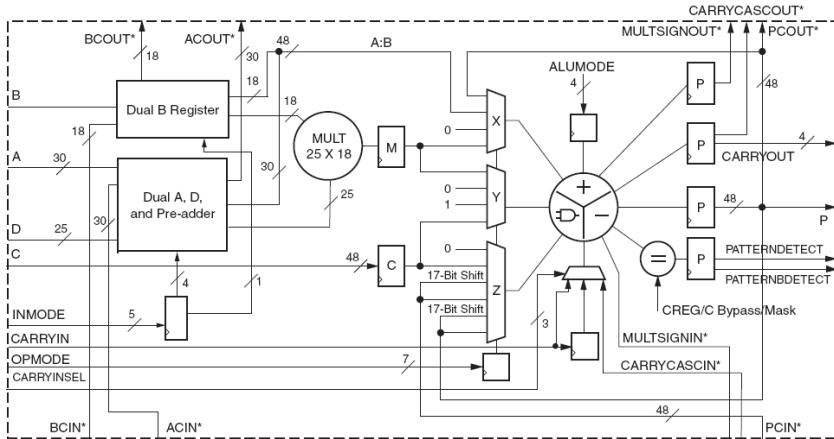
Prototyping utility: BlockRAMs are major building blocks, inferred automatically by synthesis tools. At the time of writing, however, FIFO logic is not automatically supported by synthesis tools but can be included via core instantiation. For more details on memory implementation, refer to chapter 7.

3.1.4. FPGA DSP resources

SoC designs often contain arithmetic functions, such as multipliers, accumulators and other DSP logic. High-end FPGAs, such as the Xilinx® Virtex-6 devices address these needs by providing a finite number of dedicated DSP blocks; in the Virtex-6

family these are called DSP48E1 blocks. These are dedicated, configurable and low-power DSP slices combining high speed with small size, while retaining system design flexibility. Figure 23 shows a block diagram of the DSP48E1 in detail.

Figure 23: Xilinx® Virtex®-6 DSP48E1 Slice



Copyright © 2011 Xilinx, Inc.

As shown in the block diagram, each DSP48E1 slice consists of a dedicated 25×18 bit two's-complement multiplier and a 48-bit accumulator, both capable of operating at 600 MHz throughput. The multiplier can be dynamically bypassed, and two 48-bit inputs can feed a single-instruction-multiple-data (SIMD) arithmetic unit (dual 24-bit add/subtract/accumulate or quad 12-bit add/subtract/accumulate), or a logic unit that can generate any one of 10 different logic functions of the two operands.

The DSP48E1 includes an additional pre-adder, typically used in symmetrical filters. This feature improves performance in densely packed designs and helps reduce the number of logic slices that would be required before or after the DSP block to complete a specific topology.

The DSP48E1 slice provides extensive pipelining and extension capabilities such as wide dynamic bus shifters, memory address generators, wide bus multiplexers, and memory-mapped IO register files. The accumulator can also be used as a synchronous up/down counter. The multiplier can perform logic functions (AND, OR) and barrel shifting.

Prototyping utility: very high, a major building block. Most features are inferred automatically by synthesis tools, with the exception of pattern detect which can be included via core instantiation.

3.1.5. FPGA clocking resources

Clock resources and clock networks are a major differentiator between the FPGA and SoC technologies. Whereas SoC designers have almost complete freedom to specify as many clock networks as they can imagine of many varieties, there is a real and finite limit on how many of these can be implemented in an FPGA. The mapping of SoC clocks into FPGA clock resources can be the cause of significant project delays if not catered for properly by the SoC team, for example by providing a simplified version of the SoC clocking.

Prototyping is best performed on the RTL of the design before any clock tree synthesis, before tree segmentation for test and before clock manipulation for power reduction. Nevertheless, even the raw RTL from an SoC design may include some very sophisticated clock networks and the FPGA device will need to handle these. Indeed, in some designs, it is the finite number of clock resources in an FPGA that is the limiting factor, rather than the device capacity or performance. It is therefore necessary to find a way to match FPGA clock resources to those of the SoC. This may be achieved by simplifying the original clock network (see Design-for-Prototyping recommendations in chapter 9) or by maximizing use of available FPGA clocks (see chapter 7).

Clocking resources can be divided into clock generation and clock distribution.

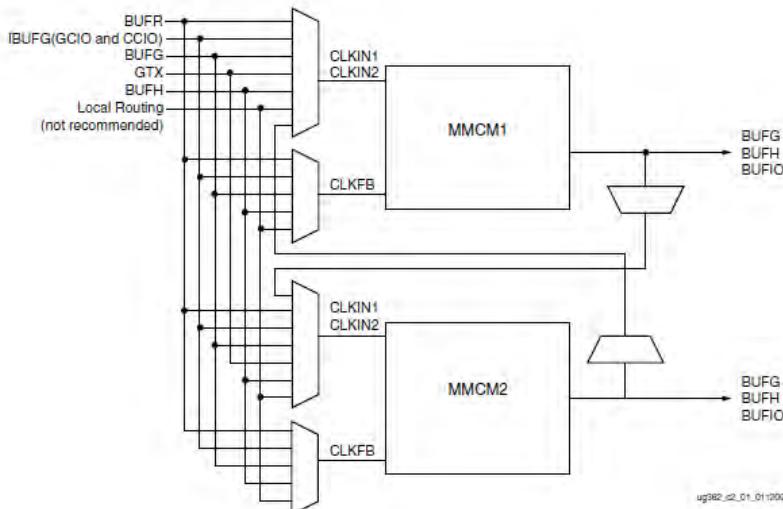
3.1.5.1. FPGA clock generation

Clocks are generated in configurable unit called CMT (clock management tile) which, in the Xilinx® Virtex-6 family, includes two mixed-mode clock managers (MMCMs). The MMCM is a multi-output frequency synthesizer based around a phase-locked loop (PLL) architecture with enhanced functions and capabilities. Each MMCM within the CMT can be treated separately; however, there exists a dedicated routing between MMCMs to allow more complex frequency synthesis. A diagram of the CMT is shown in Figure 24, giving a high-level view of the connection between the various clock input sources and the MMCM-to-MMCM connections.

The voltage controlled oscillator (VCO) in the PLL is capable of running in the 400MHz to 1600MHz range and minimum input frequency is as low as 10MHz, and has programmable frequency dividers and phase selection registers to provide output taps at 45° intervals.

Other programmable features include PLL bandwidth selection, fractional counters in either the feedback path (enabling the PLL to act as a clock multiplier) or in one output path, and fixed or dynamic phase shift in small increments.

Figure 24: Xilinx® Virtex®-6 Clock Management Tile (CMT)



Copyright © 2011 Xilinx, Inc.

All this adds up to a very capable clock generation block and there are up to nine of these in each FPGA. We should therefore not be short of options when it comes to mapping SoC clock networks into the prototype.

Prototyping utility: very high, major building block. Many clock features are not inferred automatically and must be instantiated into an FPGA version of the RTL.

3.1.5.2. FPGA clock distribution

FPGA vendors, for decades, have put a great deal of effort into producing devices with as many clocks as possible yet without being wasteful of area resources. As a result, FPGAs are very good for implementing regular synchronous circuits with a finite number of clock networks. For efficiency, there is a hierarchy of different clock resources on most devices from global low-skew clocks down to local low-fanout clocks. Once again it is the task of synthesis and place & route to ensure good usage of these resources but also, manual intervention may be sometimes required to ease the task, as will be discussed in chapter 7.

In the case of the Xilinx® Virtex-6 family, each FPGA provides five different types of clock lines to address the different clocking requirements of high fanout, short propagation delay, and accomplish low skew across the device.

Xilinx® Virtex -6 clock distribution resources include:

- **Global clock lines:** each Virtex-6 FPGA has 32 global, high fanout clock lines that can reach every FF clock, clock enable, set/reset, as well as many logic inputs. There are 12 global clock lines within any region. Global clock lines can be driven by global clock buffers, which can also perform glitch-less clock multiplexing and the clock-enable function. Global clocks are often driven from the CMT, which can completely eliminate the basic clock distribution delay.
- **Regional clocks:** can drive all clock destinations in their region as well as the region above and below. A region is defined as any area that is 40 IOs and 40 CLBs high and half the chip wide. Virtex-6 FPGAs have between six and 18 regions. There are six regional clock tracks in every region. Each regional clock buffer can be driven from either of four clock-capable input pins and its frequency can optionally be divided by any integer from one to eight.
- **IO clocks:** especially fast clocks that serve only IO logic and serializer/deserializer (SERDES) circuits. Virtex-6 devices have a high-performance direct connection from the MMCM to the IO directly for low-jitter, high-performance interfaces.

Prototyping utility: very high, major building block, automatically inferred by synthesis tools. If regional clocks are required, then location constraints are often necessary in order to associate clock load with specific regions.

3.1.6. FPGA input and output

As we shall see later as we discuss multi-FPGA-based prototyping hardware, the ability to pass synchronous signals between FPGA devices, even to the point of multiplexing different signals onto the same wire, depends on the presence of fast and flexible IO pins and clocking resources at the FPGA boundaries. As with clocking, the finite number of IO pins can often be a more limiting factor than device capacity or internal performance.

In Xilinx® Virtex-6 devices there are 240 to 1200 IO pins depending on device and package size. Each IO pin is configurable and can comply with numerous IO standards, using up to 2.5V. With the exception of supply pins and a few dedicated configuration pins, all other package pins have the same IO capabilities, constrained only by certain banking/grouping rules.

All IO pins are organized in banks, with 40 pins per bank. Each bank has one common V_{CCO} output supply-voltage pin, which also powers certain input buffers. Some single-ended input buffers require an externally applied reference voltage

(V_{REF}). There are two V_{REF} pins per bank (except configuration bank 0). A single bank can have only one V_{REF} voltage value.

Characteristics: single-ended outputs use a conventional CMOS push/pull output structure driving high towards V_{CCO} or low towards ground, and can be put into high-Z state. In addition, the slew rate and the output strength are also programmable. The input is always active but is usually ignored while the output is active. Each pin can optionally have a weak pull-up or a weak pull-down resistor. Further details of the IO pins' single-ended operation are:

- **IO logic:** each IO pin has an associated logic block in which a number of options can be selected:
- **Configuration:** all inputs and outputs can be configured as either combinatorial or registered. Double data rate (DDR) is supported by all inputs and outputs.
- **Delay:** any input or output can be individually delayed by up to 32 increments of ~ 78 ps each. This is implemented as IODELAY. The number of delay steps can be set by configuration and can also be incremented or decremented dynamically while in use. IODELAY works with a frequency close to 200MHz. Each 32-tap total IODELAY is controlled by that frequency, thus unaffected by temperature, supply voltage, and processing variations.
- **Drive current:** the FPGAs might be required to interface to a wide variety of peripherals, some mounted on daughter cards that have yet to be created. Virtex-6 FPGA IO pins can be configured to support different drive strengths from 2mA up to 24mA .

Any pair of IO pins can be configured as differential input pair or output pair. Differential input pin pairs can optionally be terminated with a 100Ω internal resistor. All Xilinx® Virtex-6 devices support differential standards beyond LVDS: HT, RSDS, BLVDS, differential SSTL, and differential HSTL.

- **ISERDES and OSERDES:** SERDES blocks reside inside the IO structure. Each input has access to its own deserializer (serial-to-parallel converter) with programmable parallel width of 2, 3, 4, 5, 6, 7, 8, or 10 bits and each output has access to its own serializer (parallel-to-serial converter) with programmable parallel width of up to 8-bits wide for single data rate (SDR), or up to 10-bits wide for double data rate (DDR). We shall see in chapter 8 how the SERDES blocks can be used to great effect in enabling high-speed time-division multiplexing of signals between FPGAs.

There are other more complex IO blocks, such as gigabit transceivers and PCIe blocks and there are references in the bibliography where the reader can find out more about using these blocks for specific purposes in an FPGA-based prototype.

There is also discussion in chapter 10 about the use of built-in IP in the FPGA to mimic the IP in the SoC under test.

Prototyping utility: IOs are major building blocks for design top-level IO and for inter-FPGA connection. Default single ended and DDR IOs are automatically inferred. Different IO types are selected by attributes assignments in the synthesis constraint manager and then are passed to the place and route tools. IODELAYs, and IO SERDES can be included only via core instantiation.

3.1.7. Gigabit transceivers

Increasingly common in ASIC and SoC designs are fast serial communication channels, used to connect ICs over a backplane, or over longer distances. These are always instantiated as black boxes in the RTL design with references to physical IO elements in the final silicon layout. We shall see in later chapters how this might be handled in an FPGA-based prototype. To model these ultra-fast serial transceivers in an FPGA requires specialized and dedicated on-chip circuitry, including differential IO capable of coping with the signal integrity issues at these high data rates.

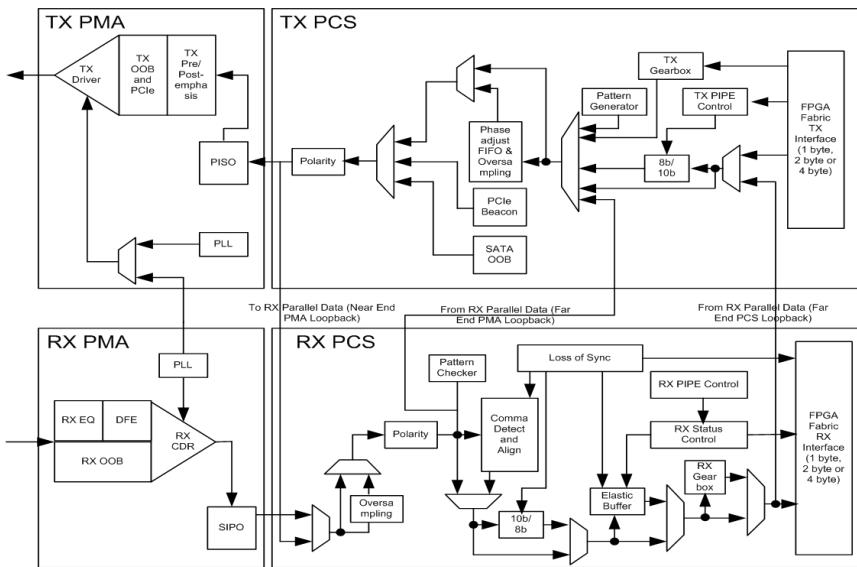
In the Xilinx® Virtex-6 family this high-speed serial IO is enabled by the presence of gigabit transceiver blocks, or GTX blocks for short. A detailed schematic of a GTX block is shown in Figure 25, which shows that as well as the physical transmit and receive buffers, the GTX blocks also have the ability to implement the physical media attachment (PMA) and physical coding sub-layer (PCS). Each GTX transceiver also has a large number of user-configurable features and parameters.

Each transceiver is a combined transmitter and receiver capable of operating at a data rate between 155Mb/s and 6.5Gb/s. Both the transmitter and receiver are independent circuits that use separate PLLs to multiply the reference frequency input by certain programmable numbers between two and 25, to become the bit-serial data clock.

Considering first the transmitter, this is fundamentally a parallel-to-serial converter with a conversion ratio of 8, 10, 16, 20, 32, or 40. The transmitter output drives the PC board with a single-channel differential current-mode logic (CML) output signal.

In its turn, the receiver is fundamentally a serial-to-parallel converter, converting the incoming bit-serial differential signal into a parallel stream of words, each 8, 10, 16, 20, 32, or 40-bits wide. The receiver takes the incoming differential data stream, feeds it through a programmable equalizer – to compensate for PC board and other interconnect characteristics – and uses the F_{REF} input to initiate clock recognition.

Figure 25: Xilinx® Virtex® -6 GTX block schematic



Copyright © 2011 Xilinx, Inc.

The different members of the Xilinx® Virtex-6 family have between 8 and 36 (GTX) circuits each with the exception of the largest device, the LX760, which does not have GTX capability. Therefore, if we need to prototype high-speed serial IP in our designs then some mix of FPGAs may be required and we shall explore this in chapter 5.

Prototyping utility: while a very powerful capability, due to their complexities and configurable options, GTX and their associated logic are not automatically inferred by synthesis tools. These blocks however, can be included via core instantiation.

3.1.8. Built-in IP (Ethernet, PCI Express®, CPU etc.)

Typically, many networking and communications SoC designs include Ethernet or PCI Express channels, so how would these be modeled in an FPGA-based prototype? FPGAs become ever more capable in their ability to implement standard interfaces at the MAC (Media Access Controller) and PHY (Physical interface transceiver). If the chosen FPGA has these capabilities built-in, then these can be used to substitute for those physical IP blocks which will eventually be embedded in the SoC but which probably appear as black boxes in the RTL. There is more information on this substitution in chapter 10.

In the case of Virtex-6 FPGAs, PCIe and Ethernet MAC and PHY are integrated into the FPGA fabric.

For Ethernet, there are up to four tri-Mode (10/100/1000 Mb/s) Ethernet MAC (TEMAC) blocks designed to the IEEE Std 802.3-2005. These can be connected to the FPGA logic, the GTX transceivers, and the IO resources and support speeds up to 2.5Gbit/sec.

For PCIe, all Xilinx® Virtex-6 LXT and SXT devices include an integrated interface block for PCI Express technology that can be configured as an endpoint or root port, designed to the PCIe base specification revision 2.0. This block is highly configurable to system design requirements and can operate 1, 2, 4, or 8 lanes at the 2.5Gbit/s data rate and the 5.0Gbit/s data rate.

Some FPGAs include CPU hard cores, often an ARM IP core of some kind. These are optimized for FPGA and will run at much higher speed than the RTL of the SoC equivalent when synthesized into FPGA, often by a factor of 10:1. Their usefulness for prototyping can be very high but only if the FPGA's core matches the actual cores built into the SoC. In addition, most SoC designs today are running multiple CPU cores, often with different capabilities or configurations. With a single, hard CPU core in each FPGA, the partitioning criteria will be driven by the need to split the design with one CPU in each FPGA. This may not be ideal for running the bus and other common design elements so the advantage gained in CPU speed may not translate to significant gain in overall prototype speed.

If there is much compromise in replacing the SoC core with a limited subset in the FPGA, then we might be better off using an external test chip or compromising on CPU speed rather than functionality. There is more discussion on IP in prototyping in chapter 10. In any case, if the CPU core is available in the largest FPGA in the family so that we do no compromise total resources, then it does not harm us to have the CPU present and we might be able to use it in a future design. In some cases, where Design-for-Prototyping procedures have been adopted by a team, the SoC CPU might even be chosen because it has a very close equivalent available in a FPGA. Our manifesto for Design-for-Prototyping procedures, of which this is admittedly an extreme example, is included in chapter 9.

Prototyping utility: while very powerful capability, hard IP blocks are not automatically inferred by synthesis tools. These blocks however, can be included via core instantiation as replacement for SoC blocks.

3.1.9. System monitor

Prototype designs can exercise a large proportion of an FPGA at high speed, so power dissipation, heating, voltage rails etc. may come under stress, especially if the design is not performing as expected, or under the influence of a bug. As we shall

see in the chapters about choosing or building FPGA platforms, a built-in monitor of the FPGAs in the working prototype can be crucial in avoiding damage due to incorrect operation.

Each Xilinx® Virtex-6 FPGA contains a system monitor circuit providing thermal and power supply status information. Sensor outputs are digitized by a 10-bit 200k sample-per-second analog-to-digital converter (ADC). This ADC can also be used to digitize up to 17 external analog input channels. The system monitor ADC utilizes an on-chip reference circuit. In addition, on-chip temperature and power supplies are monitored with a measurement accuracy of $\pm 4^{\circ}\text{C}$ and $\pm 1\%$ respectively.

By default, the system monitor continuously digitizes the output of all on-chip sensors. The most recent measurement results together with maximum and minimum readings are stored in dedicated registers for access at any time through the DRP or JTAG interfaces. Alarms limits can automatically indicate over temperature events and unacceptable power supply variation. A specified limit (for example: 125°C) can be used to initiate an automatic power down.

The system monitor does not require explicit instantiation in a design. Once the appropriate power supply connections are made, measurement data can be accessed at any time, even before configuration or during power down, through the JTAG test access port (TAP).

We will see in chapter 5 how the system monitor can be used on a prototype board.

Prototyping utility: this block is primarily a “house-keeping” monitor, usually external to the actual design, and typically used via the JTAG chain to read the device and system’s health. It does however offer a unique opportunity to include ADC in the design and if desired can be included via core instantiation. For more details on core instantiation, refer to Chapter 10.

3.1.10. Summary of all FPGA resource types

Before we move on to the tools and flows in FPGA-based prototyping, let us summarize the different FPGA resources that we have highlighted so far and their usefulness for prototyping.

Table 3 summarizes the different blocks found in most large-scale FPGAs today. All FPGA resources are useful or indeed they would not be there in the first place, however, they are aimed at a wide range of users who employ their FPGAs in real world production applications. Their usefulness in prototyping SoCs will depend upon the ease with which the SoC elements can be mapped into them and the compromise which may be required to do so. Let us now look closely, then, at the tools which enable us to use these FPGA resources to our best ability during an FPGA-based prototyping project.

Table 3: Summary of usefulness of various FPGA resources

Resource	Utility for prototyping	Inferred?
Logic Blocks	Very high, the essential building block	Always
RAM Blocks	High major building block	Usually
DSP Blocks	High, major building block.	Usually (some IP instantiation)
Clock Generation	Very high, an essential but limited resource	Often (may need RTL change)
Clock Distribution	Very high, global nets are precious resource	Usually
General IO	Very high, an essential and precious resource	Always (type set by attribute)
Fast Serial IO	High, useful for prototyping standard IP blocks	Seldom (requires IP instantiation)
Hard IP	Very powerful blocks but utility is design dependent.	Never (requires IP instantiation)
System Monitor	High, will protect investment in FPGA hardware	Never (requires design in)

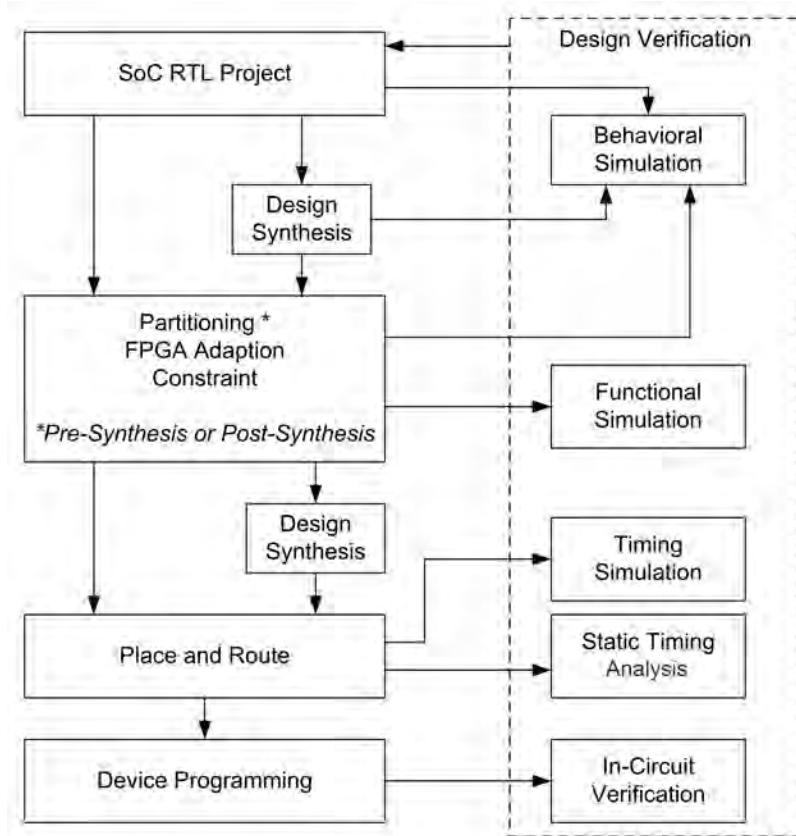
3.2. FPGA-based Prototyping process overview

Figure 26 shows the basic flow that we follow in the FPGA-based prototyping process: Let's quickly look at each of these steps in turn.

- **Synthesis:** may be performed before or after partitioning. The process of converting RTL into an FPGA netlist. The synthesis process generates an FPGA netlist for the device of choice, and the implementation constraints to be used by the FPGA back-end tools. In addition, some synthesis tools provide early estimation of the expected performance, which allows the

user to make changes to the design or constraints before spending any time in the potentially lengthy back-end process.

- **Design adaptation for FPGA:** in this step, the SoC RTL design is modified to better fit the FPGA technology and the specific prototyping platform. Typical modifications to the SoC RTL include the removal of blocks not to be prototyped, replacing some SoC-specific structures with FPGA structures like clock generation and other IP, and resizing blocks like memories to better fit in FPGA.



modified to better fit the FPGA technology and the specific prototyping platform. Typical modifications to the SoC RTL include the removal of blocks not to be prototyped, replacing some SoC-specific structures with FPGA structures like clock generation and other IP, and resizing blocks like memories to better fit in FPGA.

- **Partitioning:** the process in which the FPGA-ready version of the SoC RTL design is divided into blocks that map into individual FPGAs. This step is needed for designs that do not fit into a single FPGA. Partitioning

can be done manually or using partitioning tools. A number of approaches to partitioning were explored in chapter 3.

- **Constraint generation:** this is a convenient point in the flow to enter the various implementation constraints such as timing and pin placements. Although constraints may be generated and applied to the back-end tools after synthesis, doing so prior to the synthesis step allows synthesis to produce an FPGA netlist that is more optimized to meet the area/speed constraints after place & route.
- **Place & route:** the process of converting the FPGA netlist and the user constraints into an FPGA bit stream which will be loaded into the FPGA to provide it with the design functionality. This is often simply referred to as place & route, but in fact involves a number of steps such as mapping, place & route and timing analysis.

We will take a closer look in particular at all of the implementation steps but we do not plan to cover the verification stages in this book except to recommend that as much as possible of the existing SoC verification framework is maintained for use during the prototyping project.

At all points in the flow it is important to have ways to verify our work to that point. Re-using the original RTL testbenches and setup may require some adaptation to match the partitioned design. For example, after partitioning, a top-level netlist is required to link the partitioned FPGA netlists into a whole SoC design; often this top-level can be generated by the partitioning tools themselves.

Even if not for the whole design but for only sub-functions, maintaining a verification framework will pay us back later when we need to check functional issues seen in the design on the bench.

3.3. Implementation tools needed during prototyping

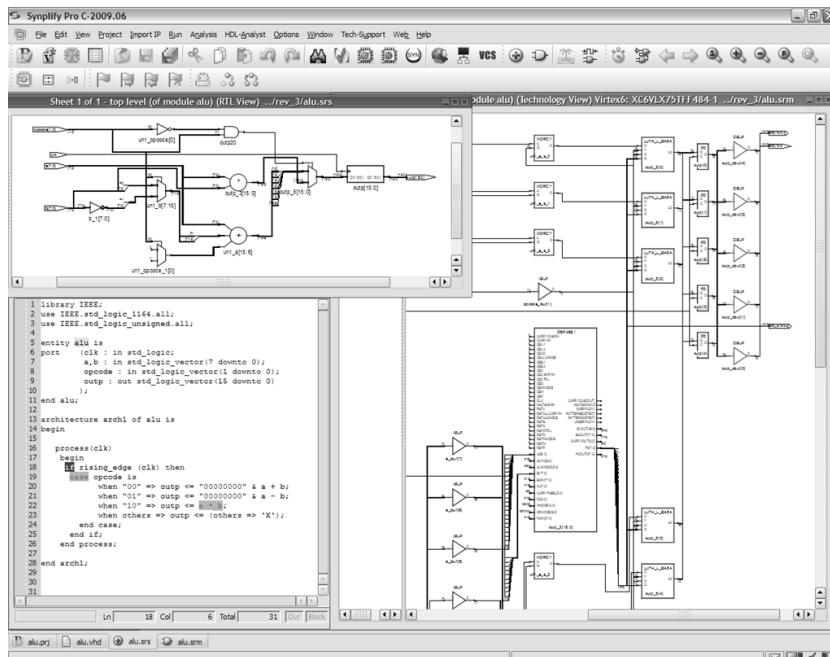
We have now explored the capability of the FPGA devices in some detail but these are of little interest if they cannot be readily employed in our prototype project. In the prototyping utility boxes above we have already mentioned that some resources are automatically employed whereas others will need some particular consideration. The ability of EDA tools for FPGA to make good use of the device resources is equally important as the resources themselves. We will now give an overview of the main EDA tools in the FPGA flow today, namely synthesis tools, partitioning tools, place & route tools and debug tools. We aim to keep the explanations as generic as possible and in each case give only small examples of tools from our own companies. More specific detail on tools available from Synopsys® and Xilinx® is available via the references. There is also more detail on the use of the tools in other chapters, particularly in chapters 7, 8 and 11.

3.3.1. Synthesis tools

As with almost every EDA tool flow, at the heart we find synthesis. For FPGA-based prototyping, we find synthesis converting the SoC RTL into a number of FPGA netlists to be used by the back-end tools, which then finally place and route the FPGA. However, at the same time the synthesis process is expected to infer regular structures from the RTL, optimize them and efficiently map them into the FPGA, meeting both space and performance goals.

As a brief illustration of this process, Figure 27 shows a screenshot of a Synopsys FPGA synthesis tool, and three views of the same small ALU design. In the bottom left is a text editor showing the RTL and the behavior extracted from that during the first stage of synthesis is shown above it. We can see a mux selecting the result of three different operations upon the inputs dependent upon an opcode and its output passing to a register. On the right of the screenshot we see a part of the final logic created by the synthesis, in particular note the use of LUTs for the multiplexing, FFs for the register and a DSP48 block used by default to implement the multiplier.

Figure 27: Synplify Pro® FPGA synthesis screenshot



In the above example, we might decide that we do not want to waste such a powerful resource as a DSP48 block to implement a simple multiplier, so we could

add an extra command, called an attribute, into the RTL or a parallel constraint file in order to override the default mapping. This is exactly the kind of control that a synthesis user has upon the way that the RTL is interpreted and implemented.

Let us a look a little more closely at the way that synthesis maps RTL to FPGA resources.

3.3.2. Mapping SoC design elements into FPGA

This section describes the tool's features that support mapping of the SoC design into an FPGA, making use of our example FPGA devices from the Xilinx® Virtex-6 family. Many of the FPGA's resources are supported transparently to the user because the synthesis tool automatically infers FPGA structures to implement user's RTL with minimal or no intervention by the user.

3.3.2.1. Logic mapping

As logic is the primary resource for logic prototyping, mapping RTL into CLBs is an elementary function. For example, for the Xilinx® Virtex-6 architecture, synthesis should be able to do the following:

- Infer LUT6 where up to six input functions are needed. LUTs will be cascaded or split when more or less inputs are needed. For example, dual LUT5s will be inferred automatically when two functions sharing up to five common inputs can occupy the same CLB.
- Memory usage in SLICEM type slices will be inferred to implement distributed RAM and fast shift registers.
- Clock enables will be inferred, with the ability to re-wire low-fanout clock enable to LUTS to maximize slice utilization.
- Set/reset, synchronous or asynchronous will be inferred including prevention/arbitration of simultaneous set/reset assertion, avoiding unpredictable behavior in silicon. For example, Synplify Pro detects such a possibility, issues a warning and then generates a logically equivalent single asynchronous reset logic.

3.3.2.2. Memory block mapping

SoC designs include many and varied memory elements and we need to map these efficiently to avoid wasting our FPGA resources. Synthesis should be able to perform the following:

- Automatically infer single and dual ported memory structures into block RAMs.
- Pack adjacent input and output registers of pipeline stages into the BlockRAMs automatically.
- Make use of BlockRAM operational modes including read-first, write-first and no-change: preserving the initial value of either of the RAM's input or output ports – as required to match the SoC's behavior.
- Automatically split larger memories beyond the capacity of a BlockRAM into multiple blocks and add the necessary logic to split and merge address and data as required. The topology of the split (i.e., optimized for speed or area) should also be controllable.

3.3.2.3. DSP block mapping

Many SoC designs include blocks which make extensive use of arithmetic and algorithmic function. If the tools can map these into the DSP blocks in the FPGA by default then a significant proportion of the FPGA resources can be liberated for other purposes.

- Adders/subtractors: FPGA logic elements have simple gate structures or configuration modes which more efficiently map carry functions enabling good implementation of basic arithmetic. Synthesis will automatically use these structures.
- Multipliers: Synplify automatically infers the use of DSP blocks for multiply and accumulate functions and operators in the RTL (see section 3.3.1 above for an example).
- Pre-adder: synthesis should infer an optional 25-bit adder before the multiplier in a DSP48 in a Xilinx® Virtex-6 device.
- DSP Cascading: for wider arithmetic in the RTL, synthesis should automatically infer multiple DSP blocks using dedicated cascading interconnect between the DSP blocks when present, for example the ports between the DSP48E blocks in a Xilinx® Virtex-6 device.

- Pipelining support: if pipeline registers are present in an arithmetic datapath then these will automatically be packed into the DSP blocks if appropriate.

As we can see above, the FPGA synthesis tools have intimate knowledge of the FPGA architecture and so as prototypers, we can rely on most of our SoC RTL being mapped automatically and efficiently without having to carve out swathes of RTL and replace it with FPGA-equivalent code.

3.3.3. Synthesis and the three “laws” of prototyping

So far we have seen that synthesis tools have the task of mapping the SoC design into available FPGA resources. The more this can be automated, the easier and faster will be the process of building an FPGA-based prototype.

Table 4: The three “laws” of prototyping

- | | |
|---------------|------------------------------|
| Law 1: | SoCs are larger than FPGAs |
| Law 2: | SoCs are faster than FPGAs |
| Law 3: | SoC designs are FPGA-hostile |

In effect, the synthesis has the task of confronting the so-called “three laws of prototyping” as seen in Table 4 below.

The clear ramifications of these “laws” are that:

- a) the design will probably need partitioning,
- b) the design may not be able to run at full SoC speed, and
- c) the design may need some rework in order to be made FPGA-ready.

Admittedly, these are really more challenges than laws and they are sometimes broken, for example, some SoC designs do indeed need only one FPGA to prototype, thus breaking the first law. However, the three laws are a good reminder of the main problems to be overcome when using FPGA-based prototyping, and of the steps required in making the design ready for FPGA.

The following sections describe the main features available in synthesis tools, with some reference to Synopsys tools, but for further information on these, please see the references.

One of the most important reasons to perform prototyping is to achieve the highest possible performance compared with other verification methods such as emulation; however, poor synthesis (or poor use of synthesis) can jeopardize this aim. It is tempting to use a quick-pass low-effort synthesis, or to reduce the target for synthesis in order to achieve faster runtime and indeed, some synthesis tools allow

for exactly this kind of trade-off. In some design blocks, however, the best possible synthesis results are essential in order to meet the overall performance target for the prototype.

The most important requirement for the synthesis is to overcome the implications of the third law of prototyping i.e., the removal or neutralization of the FPGA-hostile elements in the SoC design. Only then can we map the design efficiently into the target FPGA's resources and we will explain these fully in chapter 7.

There are a number of features of synthesis tools which are often beneficial to prototype developers. These include:

- **Fast synthesis:** a mode of operation in which the synthesis tool ignores some opportunities for complete optimization in order to complete the synthesis sooner. In this way it is possible for runtime to be made 2x – 3x faster than normal at the expense of FPGA performance. If a synthesis runtime is measured in hours, then this fast mode will save many days or weeks of waiting over the duration of a prototyping project. Fast synthesis runtime is also useful during initial partitioning and implementation trials, where only estimated design size and rough performance are required.
- **Incremental synthesis:** a feature in which the tool collaborates with the incremental implementation of the place & route tool (described below). In this mode of operation, the design is considered as blocks or sub-trees within each FPGA. The synthesis tool maintains a historical version of each sub-tree and can notice if new RTL changes impact each of the sub-trees. If the incremental synthesis recognizes that a sub-tree has not changed then it will avoid re-synthesis and instead use the historical version of that sub-tree, thus saving a great deal of time. The decisions of the incremental synthesis engine are forward annotated to the back-end place & route tools as placement constraints so that previous logic mapping and placement is maintained. A considered use of incremental synthesis can dramatically reduce the turn-around time from small design changes to final implemented design on the FPGA boards. Further details of incremental flows are given in chapter 11.
- **Physical synthesis:** a feature in which the synthesis is optimized for physical implementation where the tool accounts for actual routing delays and produces logic placement constraints to be used by the place & route tools. This feature generally yields a faster and a more accurate timing closure for the designs. This may seem contradictory to our consideration of fast synthesis above but it is often the case where one particular FPGA in a prototype struggles to reach full speed and so selective use of physical synthesis is one way that an FPGA can be brought more swiftly to timing closure.

Synthesis tools are available from third-party EDA vendors and also from FPGA vendors. We will focus on Synopsys' synthesis tools as necessary for our examples in this chapter but not to any great detail. For specific information on Synopsys FPGA synthesis, please note the references at the back of this book.

3.3.4. Gated clock mapping

One function of synthesis beyond mapping to FPGA resources is the ability to manipulate the design automatically in order to avoid RTL changes. The most significant example of this is the removal of clock gating in the SoC design in order to simplify the mapping to FPGA.

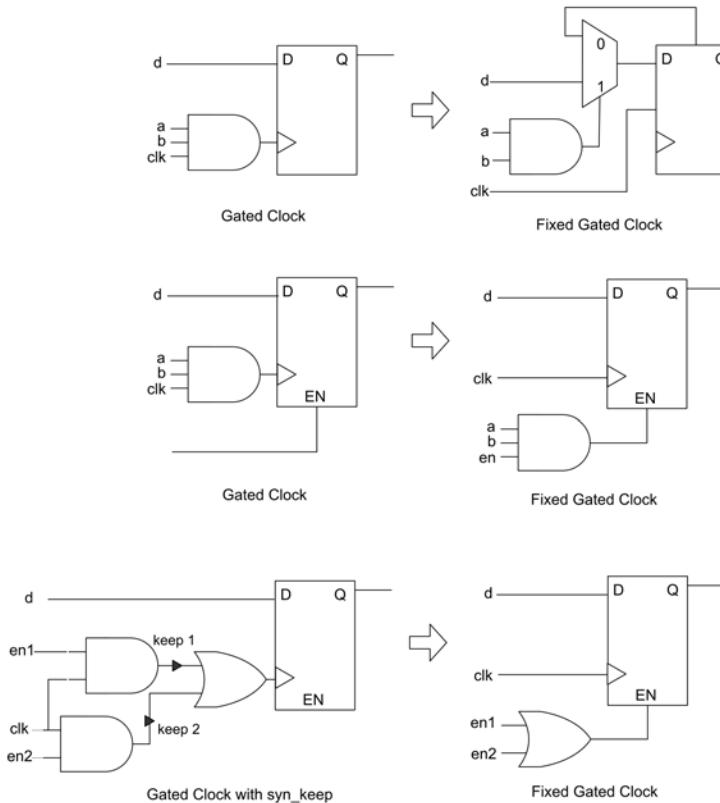
Clock gating is common in SoC designs but is not a good choice for FPGA technology where dedicated low-skew clock distribution nets deliver un-gated clock to all registers on the die. Instead of gating clocks found in the RTL, the Synopsys Synplify® tool removes the combinatorial gating from the clock nets and applies the gating logic to the clock enable pin available on most sequential elements in the FPGA.

Figure 28 shows a few examples of clock-gating translations but there will be much more description of the manipulation of gated clocks in chapter 7. Synthesis needs to be guided in which clocks to preserve, how sequential elements, including RAMs, can handle clock enables and even how black-box items can be manipulated. This is all achieved without altering the RTL.

The resulting implementation after clock gate removal is logically equivalent to the input logic, but is more resource efficient and virtually eliminates setup and hold time violations due to the low skew clock distribution.

Finally, the synthesis is only part of the flow and an important consideration is how well the synthesis can collaborate with the other parts of the flow, particularly the place & route back-end in order to ensure that all tools work towards common goals. Let's look now at the important subject of tools that perform design partitioning.

Figure 28: Examples of gated-clock removal performed by synthesis.



3.4. Design partitioning flows

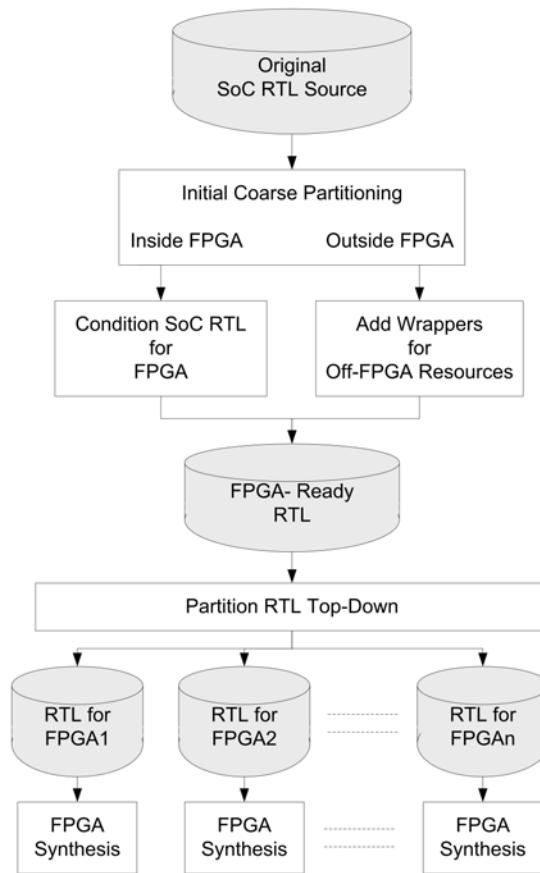
Even though FPGA capacity has increased in line with Moore's Law, the SoC designs themselves have also increased in size and complexity so SoC designs are still usually larger than today's largest FPGA devices. As a result, the first law of prototyping is as true now as it was when first proposed at the start of the millennium and the prototyper is faced with the task of partitioning the SoC design into multiple, smaller FPGA devices.

There are two main approaches to partitioning: pre-synthesis or post-synthesis. We will consider each in turn here along with a less common approach of partitioning the actual SoC netlist.

3.4.1. Pre-synthesis partitioning flow

When the partitioning is performed on the design before synthesis, the input format is the RTL of the SoC design. The partitioning task is a process of creating FPGA-sized sub-designs from the overall design tree and can be automated to some degree. Figure 29 shows the steps in a pre-synthesis partitioning tool flow.

Figure 29: Pre-synthesis partitioning flow



The flow is often performed top-down, which requires that the partitioning tools and the workstations upon which they run have the capacity to accommodate the whole SoC design, which can amount to gigabytes of data. Therefore, tool efficiency and runtime can become important factors and consideration needs to be given to the

turn-around time from RTL changes to having a new version of the design partitioned and running on the prototype board.

Originally, the pre-synthesis partitioning approach dictated that compilation and synthesis occurred on the whole design, potentially resulting in long runtimes and demanding large server resources. However, recent advances mean that the FPGA synthesis is performed on all FPGAs in parallel. This requires that the partitioning makes an estimate of final results for each FPGA in order to infer timing budgets for the IO on each device. The benefit is that total runtime is greatly improved by running multiple synthesis tools in parallel. Turn-around time for each RTL change and bug fix is reduced accordingly, especially if incremental synthesis and place & route techniques are used. More detail of incremental flows is given in chapter 11.

The drawback of this flow is that it is actually a two-pass flow. To make a correct partition, some knowledge of final FPGA resources required by each RTL module is required. If possible, and if timing-driven partitioning is our aim, then a timing knowledge at each module boundary would also be useful. This accurate knowledge of resources and timing can only come from synthesis (or preferably from place & route). We therefore need to skip ahead and pre-run synthesis before feeding back the results to the partitioner. In the Certify® tool, that is precisely what is done. The synthesis is run in a quick-pass automated mode in order to estimate resources and timing. Thus, although a two-pass flow, the pre-synthesis in effect seems like an extra “estimate” step in a single-pass flow.

The case where top-down pre-synthesis partitioning can be most powerful is when performance, especially inter-FPGA performance, is crucial. By working top-down and using the system-level constraints, a pre-synthesis partitioning flow allows for timing to be budgeted and constrained across multiple FPGAs at the same time. The synthesis is also more able to account for board-level delays and pin-multiplexing in order to correctly constrain the individual FPGAs later in their respective flows.

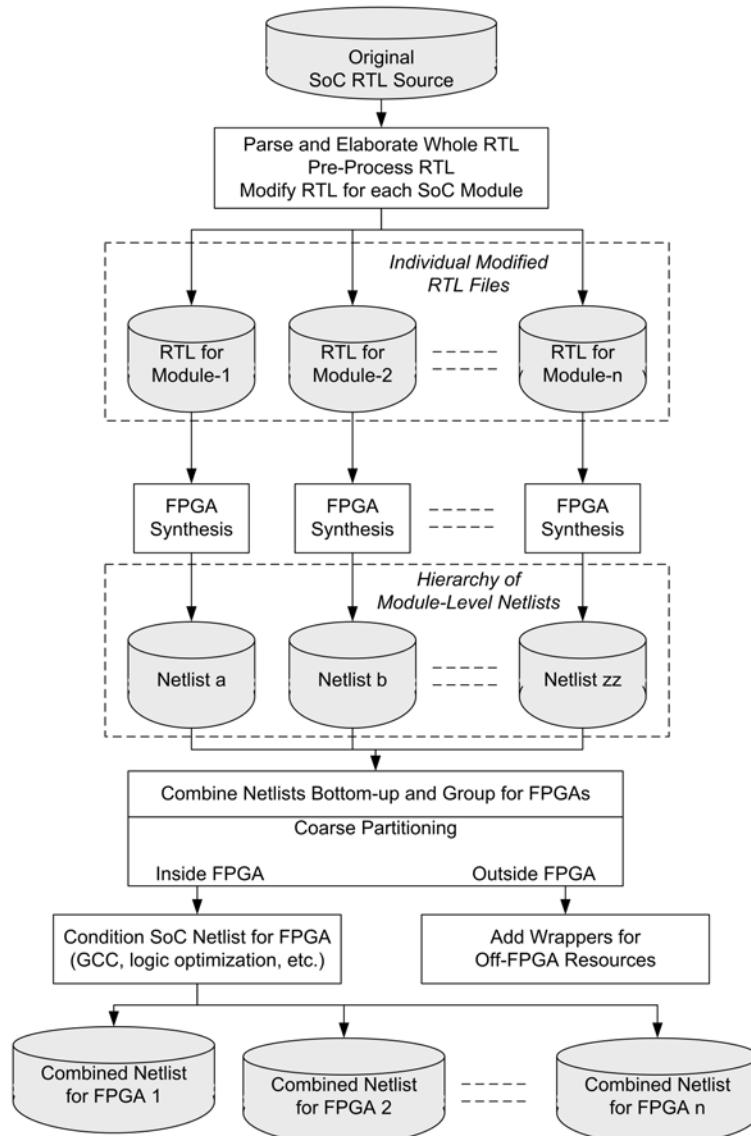
3.4.2. Post-synthesis partitioning flow

As the name suggests, post-synthesis partitioning takes place after synthesis at the netlist level. Figure 30 shows how individual modules are synthesized and mapped into FPGA elements individually, resulting in numerous gate-level netlists. The netlists are combined into a hierarchy and then re-grouped into FPGA-sized partitions. At the same time, the netlists are conditioned for FPGA (e.g., gated clocks are changed to enables) and wrappers are created for modules which will be modeled externally (e.g., RAMs). We will discuss wrappers in detail in chapter 7.

The main advantage of post-synthesis partitioning is that only those RTL source files which have changed are re-synthesized, the results of the other RTL files being adopted without change. The resultant netlists are merged and the partitioning

results are also likely to be reusable except in cases where module boundaries have altered. This lends itself to easier automation and scripting as a flow.

Figure 30: Post-synthesis partitioning flow



Another advantage of post-synthesis partitioning comes from the flow being a

Table 5: Comparing partitioning flows

	Pre-synthesis	Post-synthesis
QoR	Best	Sub-optimal
Set-up	Top-down	Simpler
Turn-around	Needs incremental synthesis and place & route	Naturally block-based
Debug advantage	Multi-FPGA instrumentation	Name preservation
Full runtime	Slightly slower	Slightly faster

natural single-pass flow. That means that by the partitioning stage, the design is already mapped into FPGA resources and timing information is accurate enough to allow more accurate partitioning decisions. There is therefore no need for a pre-run on the synthesis in order to estimate resources.

Table 5 makes a short comparison between pre-synthesis and post-synthesis partitioning flows based on the discussions above.

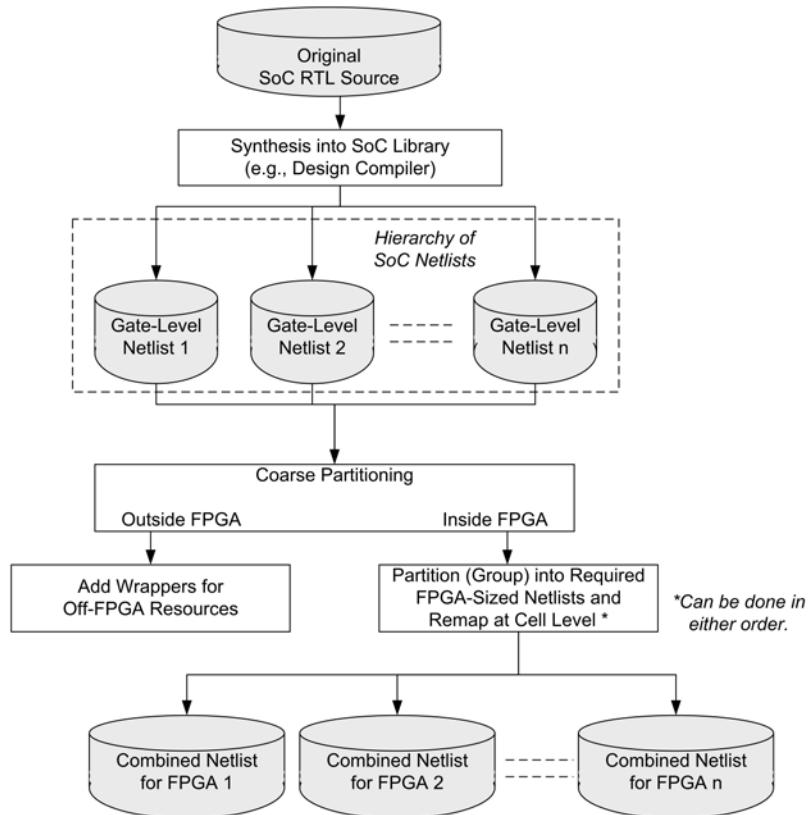
The choices are between a faster turn-around time and more automation on the one hand, and best results on the other hand.

3.4.3. Alternative netlist-based partitioning flow

There are some teams that advocate an alternative flow for FPGA-based prototyping in which the synthesis is performed by the normal SoC synthesis tools and it is the resultant gate-level netlist, or hierarchy of netlists, that becomes the input for the rest of the flow. Figure 31 shows this netlist-level flow. Here we note that normal SoC synthesis is used and the design is mapped into the same cell library as for the final SoC. The task of mapping the design into FPGA elements is performed at a cell-by-cell level, via an intermediate format where the .lib cells are replaced with their functional equivalent. During the SoC synthesis, netlists may be manipulated using built in group and ungroup style commands to do the job of partitioning. The same top-level tasks still need to be performed as in the other partitioning flows i.e.,

objects unsuitable for FPGA implementation need to be isolated and doing this at a netlist level might be too complex for many users.

Figure 31: Alternative SoC netlist-based flow



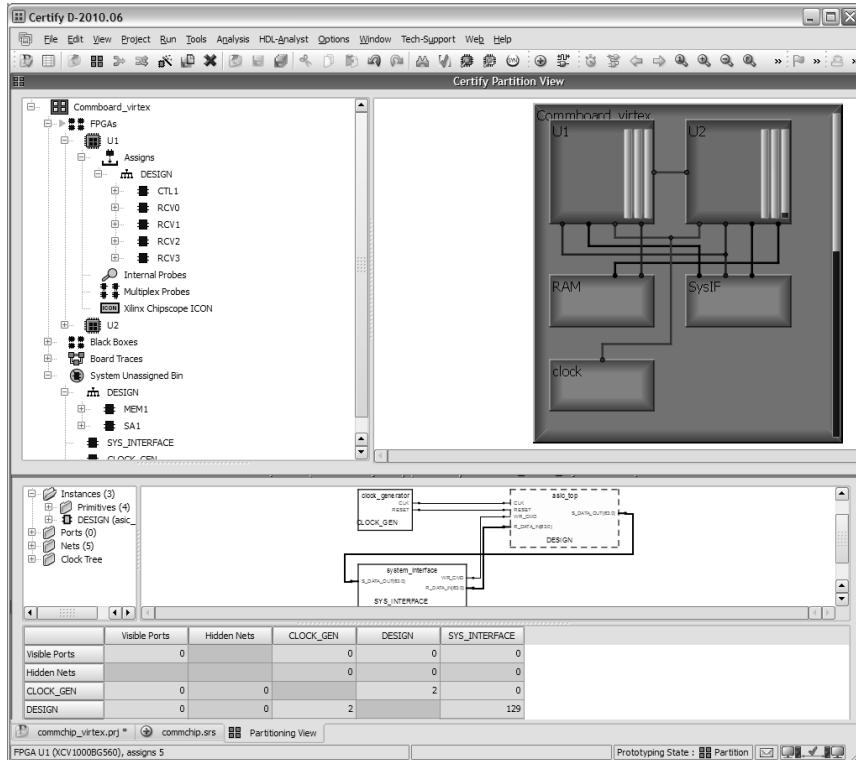
Nevertheless, netlist editors exist that allow very powerful manipulation of the design under the control of scripted netlist editor commands. Some find this preferable to changing RTL in order to do the same thing. We should understand that a netlist-based approach is likely to achieve lower performance and use more FPGA resources than the other flows because FPGA synthesis is limited to mapping a very fragmented design into FPGA low-level cells, missing many chances for optimization. All possibility of automatically inferring the high-level resources, such as DSP blocks or SRL functions of the logic elements from such a low-level netlist is lost.

We shall cover more about our uses of partitioning tools in chapter 8 but let us now move on from the front-end steps in our FPGA-based prototyping tool flow and consider the remaining steps in the flow that take our partitioned synthesized design into the FPGAs themselves.

3.4.4. Partitioning tool example: Certify®

Usually provided by third-party EDA vendors, these tools are used to automate and accelerate the partitioning of a single RTL code into multiple FPGAs. While partitioning can be done manually, for example by using group and ungroup commands and selective net-listing of subsets of the design, dedicated EDA tools significantly simplify and speed-up the initial partitioning and allow subsequent partitioning modification with ease.

Figure 32: Screenshot of Certify partitioning environment



Partitioning tools such as Synopsys' Certify, pictured in Figure 32, perform a mix of automatic, interactive (drag-and-drop) or scripted partitioning. These kinds of tools allow what-if exploration of the partitioning options which is important because some designs will not appear to have obvious partition boundaries to begin with. Tools that allow quick trials and provide immediate visibility of utilization and connectivity can guide the users to better partitioning decisions than working completely manually at the netlist level.

For example, in the Certify screen shot we can see an interactive partitioning session is in progress. The top-level of the RTL is shown schematically in the centre panel. Here we see the core of the design and a simple chip-support block alongside, in this case just clock and reset (we shall explain more about top-level partitioning in chapter 8). At the top panel, we see a representation of the board resources into which we can partition. In this case, a simple board with two FPGAs, a RAM, external IO connectors and a clock source. On the left of this panel we can also see the board's resources in a nested-list view, common to many EDA tools. In that list view and in the top-level diagram and other places we can see our progress as we assign various design elements to relevant board resources. We can also see other assignments into each FPGA, such as our debug instrumentation (in this case, Xilinx® ChipScope tools).

At each step, we get immediate feedback on how we are doing, for example, in this shot, the FPGAs have histograms showing proportion of logic, memory and IO used so far. Another useful guide for interactive partitioning is the connectivity matrix, showing the inter-block connections at this level of the design; this shot shows that there are 128 connections between the core of the design and the system-level interface (i.e., external IO).

Some further detail of the use of Certify tools, including pin multiplexing, fine-grain partitioning by logic replication and clock domain rationalization is given in chapter 7 and 8.

3.5. FPGA back-end (place & route) flow

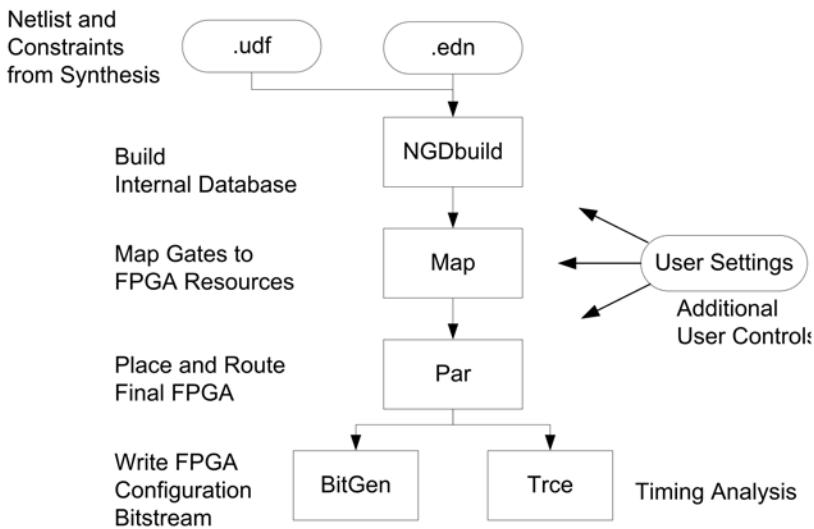
Whether or not the pre-synthesis or post-synthesis partitioning is used, the results are a mapped netlist ready for each FPGA, plus the associated constraints for timing, pin-locations etc. At the end of the tool flow is a tool or rather a set of tools, provided by the FPGA vendors which can be considered the “back-end” of the flow, using common SoC terminology. These back-end tools take the netlist and constraints provided by the synthesis tools and implement the logic into the desired FPGAs.

A simplified flow diagram of the FPGA back-end is shown in Figure 33, where we see that the first step is to re-map this netlist into the most appropriate FPGA resources as optimally as possible. If synthesis has been correctly constrained and

itself has good knowledge of the target FPGA technology then the netlist will need little remapping in the back-end.

The mapped resources are then placed into available blocks in the FPGA and routed together. The place & route runtime and results will depend on many factors, but mostly the quality of the constraints and the utilization of the device.

Figure 33: Xilinx place & route tool flow



The final step is to generate the bitstream which will be programmed into the FPGA devices themselves (with RAM-based FPGAs such as the Xilinx® Virtex® families, we call this configuration rather than programming).

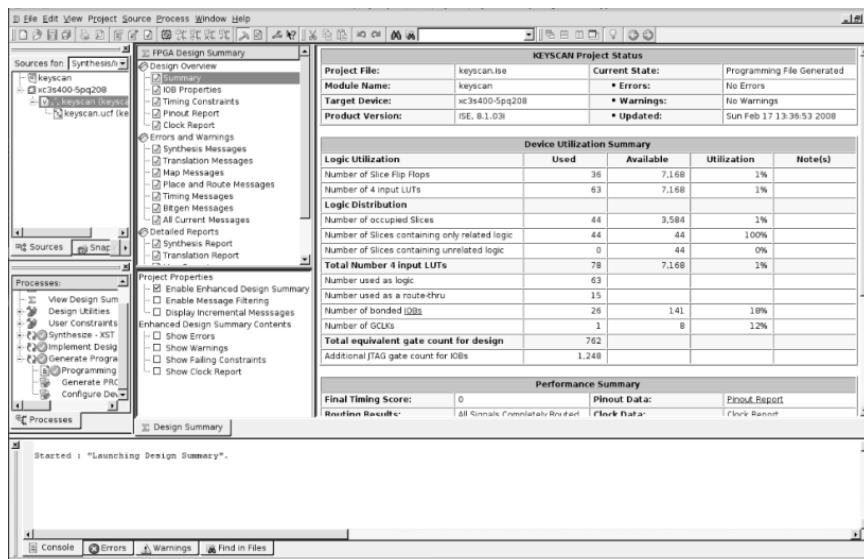
Throughout the back-end we can control and analyze the results of the various steps including timing analysis, floorplanning and even estimate the final power consumption.

These tools are generally available through a graphical design environment such as the Xilinx® Integrated Software Environment (ISE® tools) which is shown in Figure 34. Here we see the summary for a small design targeted at a Xilinx® Spartan®-3 device but the same approach is scalable up to the largest Xilinx® Virtex-6 FPGA, although the platform upon which the tool is run, especially the place & route tool, must be very much more capable. For this reason, most large prototyping projects run their tools on Linux-based workstations with maximum ram resources, which are generally widely available within SoC development labs.

Many users will run each of the above steps in an automatic flow, started by clicking a single “button” in ISE. However, it is also common to find that each step is launched individually with the appropriate arguments in a scripted flow or through a command-line interface. This allows unsupervised implementation of the FPGA in a semi-automated fashion which is beneficial when the runtime is many hours.

In that case, conditional branching or termination tests would be inserted at various points of the script to ensure that time is not wasted running tools after earlier steps had failed for any reason.

Figure 34: Xilinx® ISE® tools screenshot



Copyright © 2011 Xilinx, Inc.

A useful way to get started with place & route scripts is to use the “generate script” command in the ISE Project Navigator. This generates a tool control language (TCL) script that contains all the necessary commands to create, modify, and implement the single-FPGA project from a TCL command prompt.

3.5.1. Controlling the back-end

At the top of Figure 33, we see alongside the edif netlist (.edn) a file called .ucf. This is the user constraints file (UCF) format file which is generated by synthesis and/or manually entered by the user and which is used to control the back-end flow.

The two most important parts of the UCF are the controls for constraining the timing and the placement of the back-end results. Here is where constraints such as clock periods, IO timing, RAM placement, logic grouping and even fine-grained logic placement can be enforced on the back-end. For a FPGA-based prototyping flow, the most useful part is the placement constraint for package pins. Figure 35 shows a short excerpt of a UCF which was automatically generated by a configuration tool for the HAPS® FPGA boards, called Hapsmap. This UCF is controlling some pin locations for a Xilinx® Virtex-5 FPGA on a HAPS-51 board and is setting their voltage levels to 3.3V. The UCF is also setting a control for one of the digital clock manager (DCM) blocks as well as defining some clock constraints using TIMESPEC commands understood by the Xilinx® back-end tools. There is more information about UCF in the references and we shall take a closer look at constraining FPGA designs during chapter 7 and chapter 8.

Figure 35: Example lines extracted from typical user constraints file (UCF)

```
# Date: Tue Feb 17 16:02:12 2009
# Main board: haps-51
# Signal mapping for device A (FPGA A)

NET "CTI_HCLK_DBG"           LOC="B29" | IOSTANDARD = LVDCI_33 ;
NET "BIO1_WRITEDATA"         LOC="T36" | IOSTANDARD = LVDCI_33 ;
NET "BIO1_READ_SH"           LOC="P35" | IOSTANDARD = LVDCI_33 ;
NET "BIO1_READDATA"          LOC="T34" | IOSTANDARD = LVDCI_33 ;
NET "BIO1_CLK"                LOC="U33" | IOSTANDARD = LVDCI_33 ;
NET "BIO1_WRITE_LD"          LOC="R35" | IOSTANDARD = LVDCI_33 ;
NET "RESET_N"                  LOC="L14" ;

# DCM phase shift
INST "dcm_base_1" PHASE_SHIFT="-40";

#Begin clock constraints for 32 MHz input clock
NET "HCLK_BUFG" TNM_NET = "HCLK_intern";
TIMESPEC "TS_clk" = PERIOD "HCLK_intern" 31.000 ns HIGH 50.00%;
#End clock constraints

# Misc timing constraints
NET "CTI_HCLK_DBG_c" TNM = "cti_hclk";
NET "clk_fx" TNM = "fx_clk";
TIMESPEC "TS_01" = FROM "cti_hclk" TO "fx_clk" TIG;
TIMESPEC "TS_02" = FROM "fx_clk" TO "cti_hclk" TIG;
```

An important role for the UCF file is to act as a link between synthesis and the back-end in order to ensure that both tools are working towards the same goals. It is a common error amongst some FPGA designers to neglect the constraints for either part; for example, to provide only the most rudimentary clock constraints to the synthesis and but to then spend a great deal of time and effort tweaking controls for

place & route in order to meet timing goals. Passing UCF forward from synthesis ensures that both synthesis and place & route play their part in optimizing the design and meeting design targets.

3.5.2. Additional back-end tools

Beyond the core flow outlined above, there are a number of other useful tools in the back-end suite which may increase our productivity during our prototyping projects. These additional tools include:

- **Core generation:** a tool which generates specially constructed and optimized design elements or IP cores. Such IP cores may be part of the original design or cores that may be used to replace special RTL structures with FPGA equivalent structures. Read in chapter 10 how we can use the Xilinx® CORE Generator™ to help with SoC IP prototyping.
- **Floor planning:** a tool that allows the user to create placement constraints to any design element. Typically used for IO placements but can also be used to place logic when performance is critical and the tools cannot meet performance requirements.
- **Incremental implementation:** tools that allow incremental design implementation only on parts of the design that were changed since the last run. Depending on the extent of changes, incremental implementation can significantly reduce the implementation time, compared to a complete re-implementation. Read more about incremental flows in chapter 11.
- **FPGA editing:** an editing tool that allows modification of the FPGA after place & route. Such tools (Xilinx® FPGA Editor) allow engineers to perform low-level editing of the design (more detail in debugging tools section below).
- **In-circuit debugging:** tools that allow capturing and viewing of internal design nodes. Debugging tools will probably be used more than any other in the flow and so we will consider them in more detail next.

3.6. Debugging tools

Some thought needs to be given to how the prototype is to be used in the lab. What debug or test tools will be needed in order to provide the necessary visibility? There are a number of tools which provide debugging capabilities such as access to the design's internal nodes for probing, event triggers etc.

The debugging tools add some logic to the FPGA's design that captures selected signals into unused memory blocks based on programmable events. After capture, the signals are read out from the FPGA using the JTAG utility and are displayed in a variety of ways such as waveforms or showing values in the source code. While some tools require the user to instantiate the capture logic in the RTL prior to synthesis, other tools add the instrumentation logic at the netlist level during the FPGA implementation phase leaving the RTL intact. In addition, some of the tools allow quick tool configuration changes without needing to go through the often lengthy place & route process.

3.6.1. Design instrumentation for probing and tracing

During the prototype bring-up process, often the design does not work right away and there is a need to diagnose its state and understand its behavior. Even after the initial bring-up, during active use of the working prototype, the design may exhibit unexpected behavior. In either case, the visibility provided by the FPGA and other component IO pins is not likely to be sufficient to properly debug the situation.

The prototype debug process requires visibility beyond that available only at the IO pins and this section describes common techniques that help instrument the FPGAs that comprise the FPGA-based prototype. We will also give an overview of some tools which can greatly aid the debug process and further tool detail is offered in the appendices.

Techniques for probing internal signals of a design implemented in FPGA fall into two general categories: real-time signal probing and non-real time trace capture.

3.6.2. Real-time signal probing: test points

Viewing nodes in real-time is a common debugging practice, as in addition to signals states, real-time probing can uncover race conditions and unexpected "runt" signals. In real-time probing, signals from the design are taken to bench instruments such as logic analyzer or an oscilloscope for design analysis. Probed signals are either normally available at FPGA boundaries, or specifically brought from the design's internal nodes to FPGA test pins.

In this simplest method of probing designs' internal nodes, the user directly modifies the design and brings internal nodes to FPGA pins for real-time probing. This method consumes little or no logic resources and only a few routing resources plus, of course, the actual IO pins that bring the probed signals to the outside world.

In general, the RTL would need to be altered in order to add test points to the design, so this may simply not be an option for many projects. However, some tools support the inference of test points without changing the code. For example,

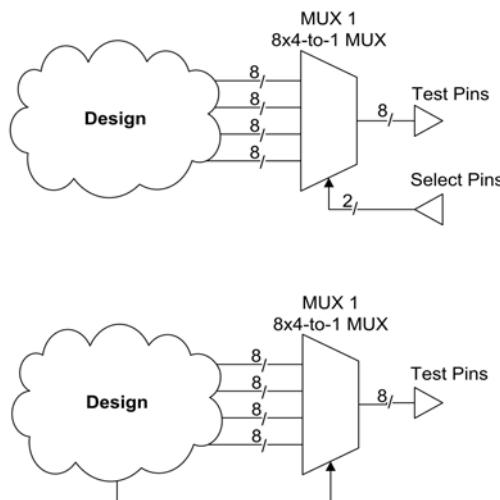
Synopsys FPGA synthesis tools support an attribute called `syn_probe` which will do exactly that.

If RTL changes do become necessary (and are allowed) then we should at least try to minimize the impact and scope of the changes. Thankfully, both major HDL languages support remote linking between different parts of a design hierarchy. This is achieved in VHDL using global signals and in Verilog-HDL using XMRs (cross module reference). More information and examples of using remote referencing is given in chapter 7.

One disadvantage of making RTL changes to bring out test points is the long turn-around time to view a different signal or set of signals. To view a new signal, the FPGA design RTL source would need to be modified and then the design is passed through the synthesis and place & route process. To mitigate this process, which might be rather long, it is recommended to consider in advance a superset of those signals that might be required for probing and bring as many to visibility points as possible. However, FPGA IO pins are a precious resource in most FPGA-based prototyping projects so the number of spare IO pins on the FPGA available to act as test points is likely to be low.

One simple and low-cost method for increasing the visibility into internal signals is to create in the design a “test header” port at the top-level to which we might connect the various signals and make changes with ease. To further minimize potential design spins or when the number of pins for signal viewing is limited, a slightly more sophisticated scheme where signals are multiplexed as shown in the following drawing:

Figure 36: Test pin muxing



As shown in Figure 36, an eight-signal wide 4-to-1 multiplexer (MUX1) is added to the design and is statically controlled with two select bits from outside the FPGA. Such a multiplexer allows probing of eight signals at the time selected from a set of 32 signals.

A second eight-signal wide 4-to-1 multiplexer (MUX2) is shown but this is controlled by an internal processor or state-machine. This arrangement saves the two select pins and simplifies the multiplexer control in some situations. If possible, we should use XMRs or global signals to connect lower-level signals for observation to the multiplexer inputs (more about the use of XMRs in chapter 7).

3.6.2.1. Note: probing using Xilinx® FPGA Editor

Another method of modifying the design quickly is to edit the design at the netlist level using FPGA tools such as the Xilinx® FPGA Editor. Using this tool the user can add design elements such as pins, and connect them to the nodes that need probing. It's important to note that a tool such as Xilinx® FPGA Editor is very powerful but complicated to use and requires a very detailed knowledge of the FPGA's resources. We therefore only recommend the use of Xilinx® FPGA Editor for experts only. There is further information in the references.

3.6.3. Real-time signal probing: non-embedded

This method of probing real-time signals is often provided as part of a commercial prototyping system, such as the CHIPit® systems from Synopsys, but could also be provided by in-house boards. The idea is to reserve a fixed number of pins from each FPGA and to route them on the board to a probing header, which acts as a debug port to which we can connect our logic analyzer. We can select the signals to be viewed on each FPGA, perhaps using a hierarchical design browser. The tool then directly modifies the already implemented FPGA netlist using one of the back-end sub-tools called Xilinx® FPGA Editor, which then connects the desired signals to the probing header.

It takes only a short time to implement the connection from internal FPGA logic to the debug port. This is possible using a tool like Xilinx® FPGA Editor because we do not need to re-run synthesis or place & route.

Care should be taken with interpreting the information given with such an approach because it is possible that signals may take widely different times to reach the debug port from their sources inside the different FPGAs. As this is all taking place on a completely implemented FPGA, the signals must take whatever paths remain available inside the FPGAs. Therefore, some of these paths will be long and via non-optimal interconnect resources, especially if the FPGA utilization is high. As a

result, the timing relationship observed between the signals will not necessarily represent the real relationship at the sources. However, it is possible to use Xilinx® FPGA Editor to measure the time delay of the path for any particular signal and then to use some logic analyzers to compensate for the known delay.

In a perfect environment, the design modification process and any timing compensation is transparent to the user but even in the lab, it is very useful to be able to quickly extract a signal and observe it on a scope.

3.6.4. Non real-time signal tracing

A shortcoming of direct FPGA signal probing is the limited number pins available for probing. A non real-time signal tracing uses a different probing mechanism and gets around this limitation. The non real-time design probing and debugging tools are available from FPGA and other EDA vendors. These tools comprise a mix of hardware and software, and provide a logic analyzer style capture in the FPGA where FPGA resources are used to implement and add to the design modules to monitor and capture a set of signals selected by the user.

Using the vendor's software tools, the user then configures the trigger condition and capture type relative to the trigger condition. Leveraging the FPGA "read back" feature, in which each FF and embedded memory in the FPGA can be read through the JTAG port. The content of the captured signal values are then transferred from the capture buffer in the FPGA to the software application running on a host computer using the FPGA JTAG pod, usually the same one used to configure the FPGA in the first place. The capture buffer content can either be displayed on the vendor's application or by other waveform display tools.

JTAG has some speed limitations, however, so some tools will make use of a faster FPGA configuration channel, if available. For example, Xilinx® devices have a facility called SelectMap which is used for fast parallel configuration. Synopsys' CHIPit debug tools use SelectMap to quickly read back the captured register information in this way.

These tools are extremely useful for an individual FPGA debug, and in addition to their use as an "after the fact" analysis, some tools have ways to cross-trigger in order to synchronize external bench tools such as signal generators or software debuggers and this helps to correlate captured events across the system.

While there are a number of similar tools available in the market place, the most common FPGA probing tools for the Virtex FPGAs are the Xilinx® ChipScope series of tools from Xilinx and the Identify® tools from Synopsys and we shall give an overview of these in the following paragraphs before moving on to a debug strategy.

3.6.5. Signal tracing at netlist level

The ChipScope tool is an FPGA debugging tool from Xilinx for general purpose use, but with some special-purpose analysis versions available, for example for linking to embedded CPUs or analyzing fast serial IO. ChipScope tool works by adding extra instrumentation resources to the design in the device for the purpose of communication, triggering and trace capture. The instrumentation is synchronous to the design and uses the same system clocks as the logic being sampled, thus avoiding setup and hold issues. It should be noted that the trace capture is in actuality a sample of activity on an internal node of the FPGA. The timing resolution is of the trace data will be only the same as the capturing clock, therefore this is not a tool for analyzing absolute timing of internal signals.

Figure 37 describes the ChipScope tool implementation flow, showing two ways to add the instrumentation items to an FPGA design.

Of these two methods, the typical implementation flow is:

- Using Xilinx's CORE Generator™ tool the user defines and generates the communication core and logic analysis cores to be embedded into the FPGA.
- The user then combines the cores with the FPGA design, either by instantiation into the RTL or directly into the synthesized design where the cores are merged at the netlist level so there is no need to modify the RTL design.
- Design must go through the synthesis place & route process before the tool is ready to be used.

The alternative is to insert the instrumentation at the netlist level:

- Use the ChipScope tool core inserter to open the netlist (either the edif from synthesis or Xilinx internal netlist, ngc).
- Select the clocks(s) and signals to be traced.
- The core inserter builds a new netlist and there is no need to use the CORE Generator™ tool.
- Pass results to place & route.
- In either case, once the instrumented design is downloaded into the FPGA, the use for debug is the same. The user communicates with the logic analysis cores via the FPGA's JTAG download cable in order to set up the trigger conditions, capture mode etc. When the trigger and capture conditions are met, the tool transfers the captured data from the FPGA and displays it in the tool's application running on the host PC.

As shown in Figure 38, multiple ILAs can be instantiated in one FPGA while using only one communication core (ICON). An interesting feature available with ILA is the ability to generate a trigger output signal. This configurable signal reflects the trigger condition and can be routed to an FPGA pin and used to trigger other bench instruments. Although there is latency of ten clocks between the trigger event and the trigger output, it can be still used to correlate the trigger events with other parts of the system.

Figure 37: ChipScope tools design flow

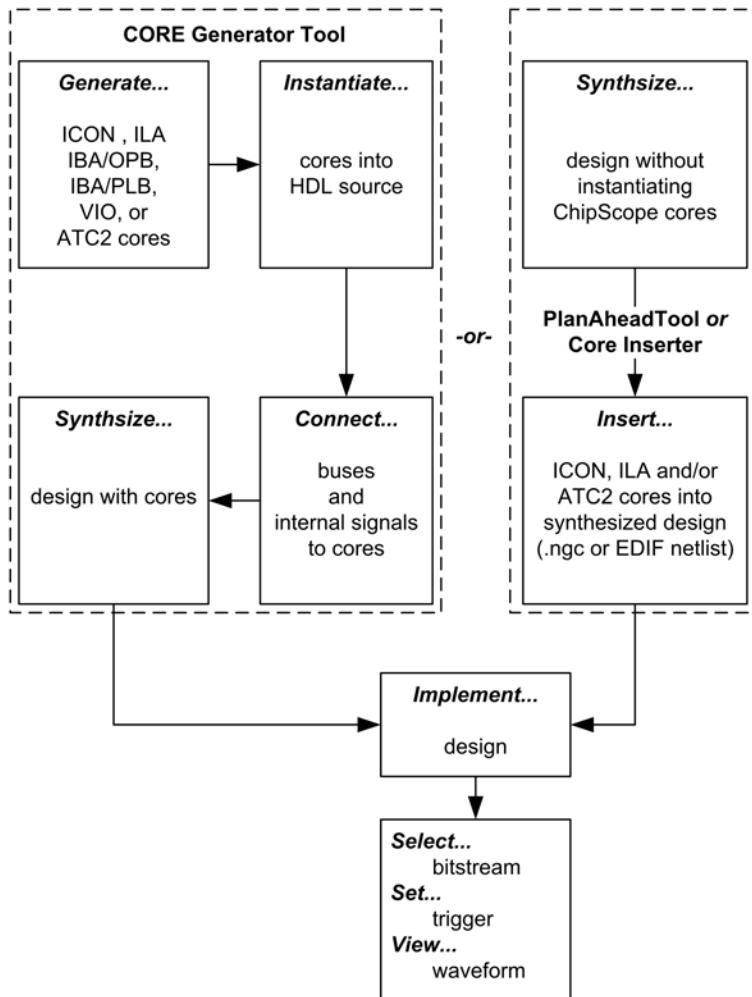
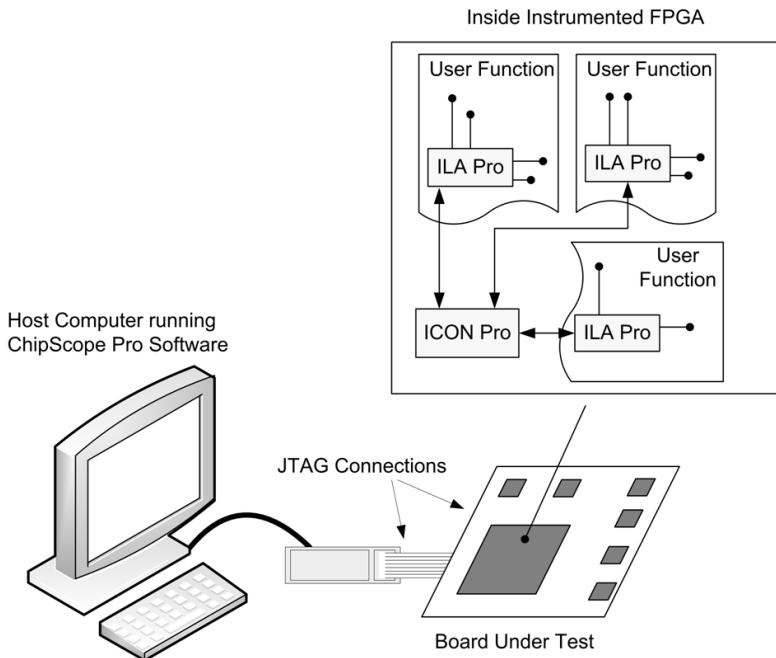


Figure 38: ChipScope™ Pro system block diagram

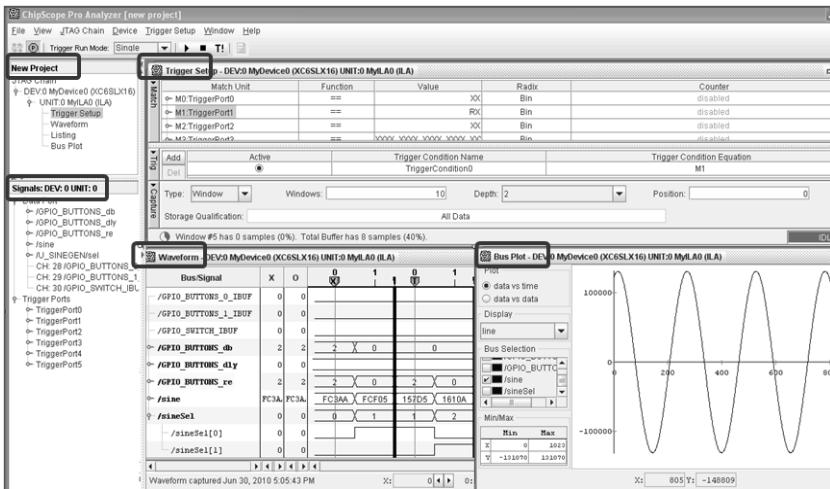


Copyright © 2011 Xilinx, Inc.

To view signals captured with LA1, Xilinx provides a custom graphical signal browser as part of the ChipScope™ software, a screenshot is shown in Figure 39.

Here we can see in the central horizontal panel an area for setting trigger values, which can be simple events or combinations of events over time using counters. There are two ways shown for viewing captured samples, one is showing a traditional logic analyzer style of view while the view to the bottom right is of the variation of a bus value over time, represented as an analog waveform. These signal viewers also allow a variety of features such as signal renaming, regrouping, reordering etc. so we are in a very familiar environment like a logic analyzer or scope, even though the “engine” of it all is embedded in the FPGA.

Figure 39: Screenshot of ChipScope™ debug tool



Copyright © 2011 Xilinx, Inc.

In addition to displaying captured data using the signal browser, ChipScope™ software can export the captured data in a variety of formats, including VCD (Value Change Dump), ASCII and Agilent's FBDF (Fast Binary Data Format). Exported data can then be further viewed or analyzed by user scripts or by a wide variety of third-party tools.

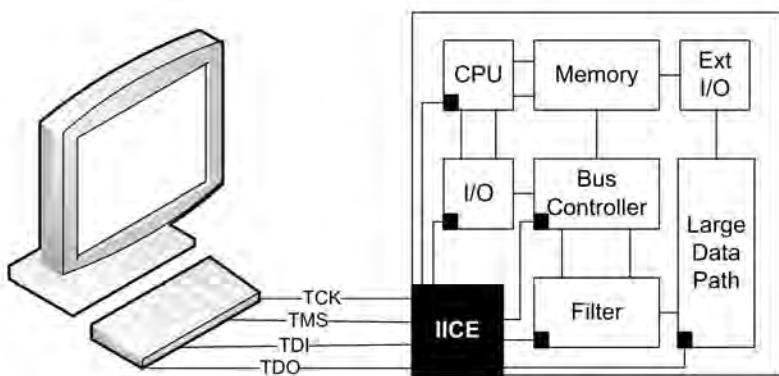
3.6.6. Signal tracing at RTL

Identify® is a tool supplied by Synopsys which works in a similar way to the Xilinx® ChipScope tool except that instrumentation takes place at the source RTL rather than at the netlist level. Like ChipScope software, Identify instruments the design by the inclusion of signal monitoring logic, trace capture and buffering, and communications to a host PC. In addition, Identify tools provide signal selection and monitoring mechanisms which enable the user to more easily trigger the trace capture and to correlate the captured information back to the source RTL.

Identify tool is comprised of two sub-tools, called the RTL Instrumentor and the RTL Debugger: The key part of the instrumentation logic itself is called an IICE (pronounced “eye-ice”) or intelligent in-circuit emulator, which is embedded into the user’s design. The IICE contains the probing logic, the trigger logic and trace buffers plus the runtime communication to the host running the RTL debugger.

Figure 40 gives an overview of the concept behind an RTL debugger. This is actually a screen shot and block diagram of the Synopsys Identify tool in its basic configuration. The tool inserts an IICE probing block that samples internal node information and gathers it together into a sample buffer, generally using one or more of the available FPGA BlockRAMs.

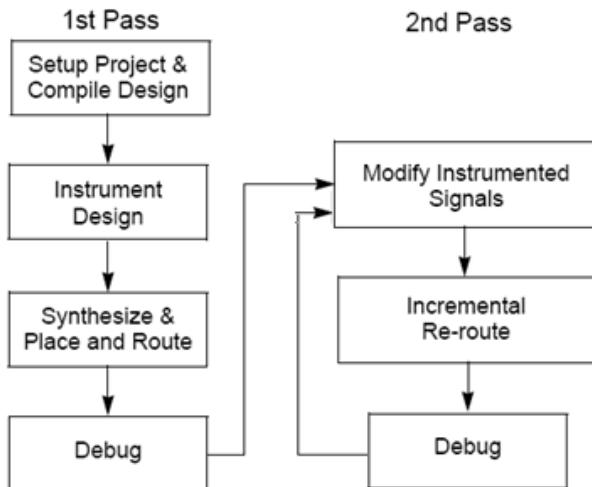
Figure 40: Synopsys Identify® tool overview



The IICE also includes triggering and sequencing logic for deciding when samples are captured. The IICE communicates over JTAG to the application where breakpoints and the state of the captured signals are shown and highlighted in the design's RTL source.

Figure 41 describes the general usage flow for Identify, showing a two-pass strategy to focus in quickly to “zoom in” on data required for a specific debug task. As shown in the left process (first pass), Identify cores are added to the design before synthesis, followed with the place & route processes, but subsequent changes (second pass) to the instrumented signals are implemented incrementally where the synthesis and place & route steps are by-passed.

Figure 41: Identify use model and flow



The following describes in more detail Identify's main usage flow:

- Using Identify's Instrumentor, the user selects signals to monitor and trigger conditions directly in the design from a hierarchical RTL source viewer.
- Instrumentation automatically generates the IICE core.
- After synthesis and place & route, the IICE core is controlled by the Identify Debugger where the user sets the trigger condition and arms and monitors the running hardware via the FPGA JTAG facility.
- After trigger, data captured in FPGA memory is uploaded to the Identify Debugger application software over the FPGA JTAG facility.
- Captured data can be displayed in a variety of ways: either directly annotated into the RTL code, or using common third-party waveform viewing tools.
- Small changes to the signal list and trigger conditions can be quickly made incrementally without going through the **synthesize place and route processes**.

A design can contain multiple IICES, each configured in different clock domains and allowing different trace and trigger conditions. IICES can cross-trigger between themselves so as to track a complex sequence of events. Multiple IICES in an FPGA share the same communication mechanism with the JTAG interface.

In addition, the IICE provides the option of exporting the trigger condition, where a copy of the trigger signal is brought to the design top level and used to sync to bench instruments or software debuggers for enhanced system-level debugging. Linking to a software debugger is covered in more detail in chapter 11.

The Identify RTL Instrumentor is a sub-tool running on the host by which the user browses the design RTL in order to select the signals to be traced and if desired added to various cones of trigger logic. During instrumentation, the IICE is

Figure 42: RTL Instrumentor source code view

```
56 begin
57     if dclr = '0' then
58         dcurrent state <= s_RESET;
59     elsif dclk'event and dclk = '1' then
60         case dcurrent state is
61             when s_RESET      => dcurrent state <= s_ONE;
62             when s_ONE       => dcurrent state <= s_TWO;
```

configured and added. The most useful aspect of making such instrumentation at the RTL is that we can easily keep track of what visibility we are adding. Figure 42 is an excerpt from an Identify screen shot which shows the RTL source code appears during an instrumentation process. The available signals for sampling are underlined and the cartoon spectacles change color if that signal is sampled, or part of the trigger logic, or both. The sphere icons on the left show where breakpoints might be inserted adding a useful feature to allow us to trap when the design reaches a certain line of code in much the same way as software engineers might debug their code.

Let us now look at the in-lab debugger part of Identify. The RTL Debugger is the sub-tool that provides an interactive interface that we use in the lab to control the triggering, capture, and various flow-control features such as breakpoints. The debugger also provides the interface to waveform viewers.

Like ChipScope software, the Identify debugger provides a number ways to view the captured sample data and signals including:

- **State annotation:** when using break points, it annotates the signals' state directly into the source code using the supplied design navigator tool. Figure 43 shows a screen shot of Identify RTL Debugger displaying the RTL source annotated with breakpoints and captured data states. In reality,

the captured data is overlaid on the source code, highlighted in yellow and users can scroll through sampled data and see the values on the source code screen change. This is roughly analogous to the stepping through a software debugging tool.

- **Waveforms:** in addition to state annotation, Identify Debugger can interface with popular waveform viewers such as the freeware GTKWave viewer or the DVE environment provided with Synopsys simulators. The trace samples can also dump the signals into a standard VCD (Value Change Dump) file that can also be displayed on a wide variety of available waveform viewers.

Figure 43: RTL debugger source code view showing sampled data

The screenshot shows an RTL debugger interface. On the left, there is a vertical toolbar with several icons. The main area displays a block of Verilog-like code. On the right, there is a panel labeled "Sampled data (in yellow)" which contains some binary values. Arrows point from specific lines of code to this panel, indicating where sampled data is being taken. The code itself includes annotations like state transitions and logic conditions.

```

Activated and triggered breakpoint
Sampled data (in yellow)

46      &grant2'0' <= '0';
47
48      case (curr_state) is
49          when st_idle1 =>
50              if ( &req1'0' = '1' ) and ( &req2'1' = '1' ) then
51                  next_state <= st_grant2;
52              elseif ( &req1'0' = '1' ) then
53                  next_state <= st_grant1;
54              elseif ( &req2'1' = '1' ) then
55                  next_state <= st_grant2;
56              else
57                  next_state <= st_idle1;
58          end if;

```

FPGA resources used by Identify consist of the capture logic and the capture buffer that is usually implemented in on FPGA BlockRAM. The size of the analysis resources depend on the number of signals to be captured, the trigger condition and to a greater extent, the desired capture buffer depth.

3.6.7. Summarizing debugging tool options

One of the traditional complaints against FPGA-based prototyping was that visibility into the design once inside FPGAs was very poor and that debug was non-intuitive because even when we had visibility, we didn't really know what we were seeing. After this section, we have seen many different ways to add debug visibility into our prototype and there are many other tools available which are variations on the instrumentation approach taken by Identify and ChipScope software. We have summarized the approaches in Table 6.

Table 6: Comparing debugging technique and tools explored in this chapter

Technique/Tool	Time span	Entry Point for instrumentation	Depth of Capture	Time to add signal
Add test points to RTL	Real-time	RTL or FPGA editor	Immediate	long
Non-embedded debug port	Real-time	Add ports directly into FPGA using editor tool	Immediate	short
Signal tracing: netlist level	Non real-time	Insertion into FPGA netlist using debug tool	1000s of 128 signal samples	medium
Signal tracing: RTL	Non real-time	Insertion into RTL using debug tool	1000s of 128 signal samples	long

3.7. Summary

In this chapter we have aimed to give an overview of FPGA devices and tools today from our unique prototyper's perspective. We described the main features of FPGAs themselves with specific reference to Xilinx® Virtex-6 family, discovering that the usefulness of each FPGA feature depends not only on its functionality but also on its support by the relevant EDA tools.

Regarding the EDA tools, we considered the need to address the so-called “three laws of prototyping” and the tools that enable us to do that, including synthesis, partitioning and place & route.

There are many more details available in our references but, now that we have a good understanding of the FPGA technologies involved, in the next chapter we shall get started in our FPGA-based prototyping project.

The authors gratefully acknowledge significant contribution to this chapter from

Joe Marceno of Synopsys, Mountain View

So far we have explored the basic technology of FPGA devices and tools. For a first-time user of FPGA-based prototyping, it may seem a long leap to get from the fine-grain details of the target technology, to understanding how it may be employed to build a prototype. Later chapters will give all the necessary low-level technical details but first, it may be useful to answer the question “where do I start?” This chapter is intended to do precisely that. We will first look at the steps required and help the reader to ascertain the necessary resources, time and tooling required to get from first steps to a completed prototype project. At the end of this chapter, the reader should be able to know whether or not FPGA-based prototyping of an SoC design can be successful.

4.1. A getting-started checklist

To set the scene, Table 7 has a framework of the main tasks and considerations during the project, broken into two main phases; set-up phase and the usage phase.

It may be obvious that most of the effort comes during the set-up phase but most of the benefit is reaped during the use phase.

The steps in the checklist follow the same order as the chapters and subjects in this book and both reflect the decision tree through which the reader should already be traversing. In FPGA-based prototyping, the SoC design is mapped into a single or multiple FPGAs, emulating the SoC behavior at, or close to, the intended SoC speed. Each subject area will be explored in detail later, but first, let us look more closely at the tool and process flow at the heart of an FPGA-based prototyping project.

Table 7: A getting-started checklist

Phase	Tasks	Considerations
Set-Up	Choose FPGAs	How much capacity, speed, IO?
	Build boards or buy boards	What expertise exists in-house? How much time is there? Does total cost fit in the budget?
	Get the design into FPGA	The “three laws of prototyping” How to minimize design changes How to track builds and revisions
Usage	Bring-up prototype and find bugs	What instrumentation do we need? Can prototypes be shared? Is remote debug or configuration needed?
	Run software	Will the software team use it standalone? How to link with software debugger?
	Run in real world	How to ensure portability and robustness? Any special encasement needs?

4.2. Estimating the required resources: FPGAs

The authors have seen situations in the past where novice prototypers have badly underestimated the FPGA resources required to map their designs. As a result, the platform, which had been designed and built in house, had to be scrapped, resulting in many weeks of delay in the project. This is an extreme example, but even if we do not overflow our available resources, we do not want to find ourselves in a situation where the FPGAs are too full, resulting in longer runtimes and lower performance.

At the start of the project, we need to establish what resources we need to implement the design and then allow for overhead. It is nevertheless important to establish an early and relatively accurate estimate for the required FPGA and other resources that we shall need. First, however, we need to decide how much of the SoC design will be prototyped and for this, we need to have early access to the SoC design itself. But how mature does the SoC design need to be in order to be useful?

4.2.1. How mature does the SoC design need to be?

We would like to have our FPGA-based prototype ready as soon in the project as possible in order to maximize the benefit to our end users e.g., software team and IP verification people. However, it is quite possible to start our prototyping effort too early and be wasting our time tackling RTL code which is too immature or incomplete. It is not an efficient use of prototyping time, skill and equipment only to find obvious RTL errors which are more easily found using RTL simulation. So how do we decide when is the optimum time to start our prototype?

There are probably as many answers to that question as there are prototype teams in the world and each team may have developed a procedure which they believe best suits their needs. For example, the prototyping team lead by David Stoller at TI in Dallas, USA, requires that the project RTL is complete and passing tests at a coverage of 80% or more before the RTL will be placed into a prototype. There are others who advocate that preliminary deliveries of RTL are usable when the code is only 75% complete. Whatever metric is used, in each case it is recommended that the release of RTL for prototyping is built into the overall SoC verification plan, like any other milestone.

All SoC teams will hold a plan for verification of all aspects of the design at various stages of the project. The aim of the plan is always 100% verification of the SoC and self-confirmation that this has been achieved. Some verification teams follow sophisticated methodology, such as the UVM (universal verification methodology), which is primarily intended for use with complex test-environments written in System Verilog. A verification plan records where verification engineers have decided to make certain tests and also the current state of the design with respect to each test. It will include various milestones for the maturity of the project and especially its RTL. For good integration of FPGA-based prototyping into the whole SoC project, the verification plan should also include milestones for the prototype itself. Basic milestones might be when RTL is ready for check-out, when the FPGA-ready RTL has passed the original test bench or when basic register access on the board is running etc. Integration of FPGA-based prototyping into verification plans is part of Design-for-Prototyping, which is covered in chapter 9.

To come back to our question of when the RTL is ready for prototyping, a good general guide is to sync the first delivery of RTL with the point in the SoC project where trial implementation of the ASIC flow is started. This is typically where the RTL has been tested at each block level and has been assembled enough to run some basic, early tests. Some teams call this the “hello world” test i.e., the RTL is assembled into a simulator and a test bench is able to run vectors in order to load and read certain registers, see ports and pass data around the SoC etc. If the design is passing such tests in the simulator then it should also be at least able to pass this in the prototype, which is a good indication that the prototype is ready to make its contribution to the validation effort. For more information about use of “hello world” tests and prototype bring-up, see chapter 11.

4.2.2. How much of the design should be included?

Early in the prototyping effort, we need to establish the scope of the prototyping. The considerations are usually a mix of the architecture and scale of the design to be prototyped, and the available prototyping options. Before we consider the partitioning between multiple FPGAs we need to apportion which parts of the SoC design will be placed into FPGA and which may be better placed elsewhere, or even left out altogether.

Figure 44: Coarse partition of design into FPGA or external devices.

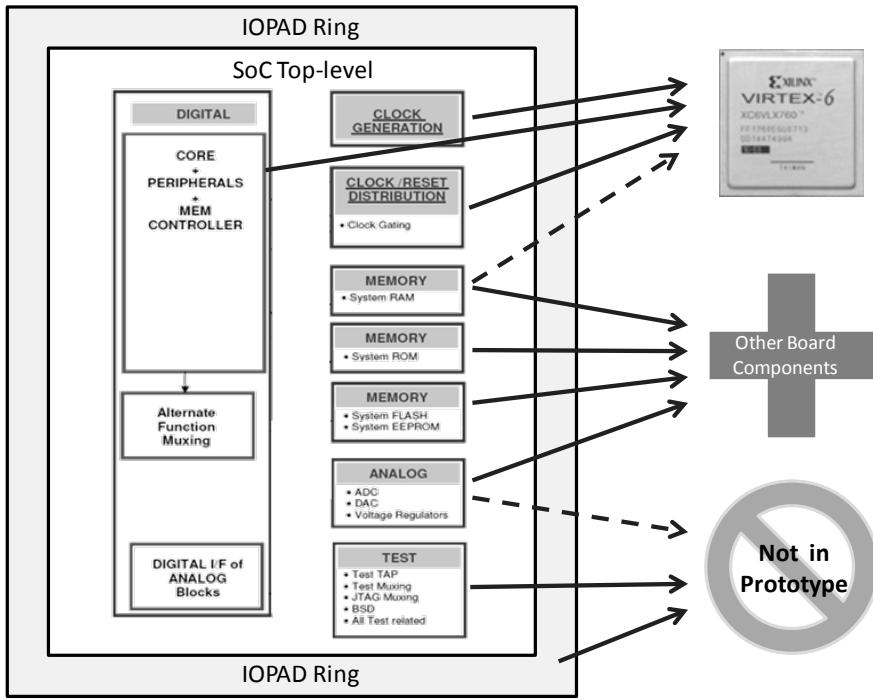


Figure 44 shows an overview of the top level of a typical SoC. Most of the SoC digital blocks can be placed inside FPGA resources but other blocks will be better placed in external resources, particularly memories. The analog sections of the SoC might be modeled on-board but some may choose to leave the analog out and prototype only the digital portion. The IO Pad ring is normally left out from the prototype along with the test circuitry as the prototype is focused upon functional test of the digital RTL and creating a working model for the software, which does not typically interface with the test system.

When evaluating the design intended for prototyping, the design can generally divided into two categories: what can go into FPGA, and what cannot go into FPGA.

4.2.3. Design blocks that map outside of the FPGA

While FPGA is the main prototyping resource in a typical prototyping system, typical SoCs are likely to have blocks that either do not map into FPGA, or blocks for which better prototyping resources are available. Such blocks are typically either analog circuits or fixed digital IP blocks for which neither source code nor FPGA netlist is available. In either case, we need to consider resources other than FPGA to prototype them.

For evaluation purposes, IP suppliers typically provide evaluation boards with the IP implemented in fixed silicon. In other cases, the prototyping team may design and build boards that are functionally equivalent to the IP blocks that do not map into FPGA. In yet other cases, existing or legacy SoCs may be available on boards as part of the prototyping project and can be added to and augment the FPGA platform. The advantage of using such hard IP resources is their higher performance as compared to FPGA implementation and their minor FPGA footprint.

To accommodate for the inclusion of these external hardware options, we need to verify/perform the following:

- **Electrical connectivity:** provide connectivity in the RTL to modules that are external to the FPGA.
- **Signaling:** make sure the IO signaling levels between FPGA and external hardware modules are compatible with each other. Also, make sure the signal integrity of the external hardware is acceptable to both FPGA and hardware module. Typical FPGA IOs have a number of options such as signaling levels, edge speed and drive strengths. We should be familiar with these options and how to best use them in the particular prototyping environment.
- **Clocking and timing:** provide the needed clocking, account for the clock delays and calculate the interconnect timing between FPGA system and external hardware.
- **Logic synchronization:** cold start-up time of fixed hardware is likely to be much shorter than that of the FPGA system as the FPGAs need to be loaded before they become functional. Early start can result in synchronization problems between the hardware modules and the FPGA system. We need to understand these issues and account for synchronous and asynchronous resets, and pulse durations for all cold and warm restarting conditions.

- **Physical/mechanical:** plan how the external blocks will physically connect to the FPGA system and how power is supplied. Care must be paid to cabling and repeated connection to the FPGA system to protect both the FPGA and the hardware module. A good mechanical plan will make it easier when the prototyping needs to be shipped to a remote lab or a customer.

Clearly only design elements that can map into FPGA may be included so, for example, analog elements will need to be handled separately. When evaluating the scale of prototyping there are some trade-offs that must be considered. Specifically, design size, clock rates and design processing times are inter-related and will mutually affect each other. While there are examples to the contrary, for a given platform, smaller designs generally run at faster clock rates and take less time to process than larger designs. We should therefore be wary of including more of the SoC design into the prototype than is useful. In which ways can we reduce the scale of the prototype without overly reducing its usefulness or accuracy?

To reduce the size of the design and to ease the capacity limitations, the following options can be considered:

- **Permanent logic removal:** when blocks in the SoC design have a hardware equivalent (physical SoC, evaluation platform etc.) that can be added to the FPGA platform, then it is a good candidate for removal from the logic that will map into the FPGA. In addition to space relief, these blocks can run at faster speeds than the attainable speed in the FPGA platform. Typical examples for such blocks are third-party IPs such as a processor, or special purpose controller.
- **Temporary logic removal:** when the amount of logic exceeds the feasible space in the FPGA platform, temporary removal of some blocks may be considered. This can work when parts of the SoC can be prototyped at the time. For example, if an SoC has multiple possible memory interfaces but only one will be used in the actual application, then prototyping them one at a time, while the others are removed, will verify each one in isolation of the other.
- **Block scaling:** when the size of blocks in the SoC will exceed the available feasible space in the whole FPGA platform, scaling them down may be considered. For example, SoC memory block may be scaled down to run subsets of the code at any given time, or alternatively, one channel of a multichannel design might be sufficient to represent enough functionality for the purposes of prototyping the channel-driver software.

Each of these recommendations is discussed in greater detail in chapter 7

Once the portion of the design to be prototyped has been selected, it is important to estimate up-front how well the remaining design will fit into FPGA technology. To do this estimation, it is obviously important to understand the FPGA's resources

available in our prototype platform. A too-simple early estimate that such-and-such FPGA contains n million gates will often lead to difficulties later in the project when it appears that the design is taking more FPGA resources than expected.

4.2.4. How big is an FPGA?

As covered in detail in chapter 3, an FPGA is an excellent resource for prototyping. Not only does it contain copious amounts of combinatorial and sequential logic but it also contains additional resources such as various types of memory and arithmetic blocks, and special IO and interconnect elements that further increase the scope of FPGA technology for prototyping.

The heterogeneous nature of modern FPGAs makes the answer to the question “how big is an FPGA?” rather difficult. The generic answer might be “it depends.” We will give some idea of the raw capacity of various parts of an FPGA below, but in fact, their use in an FPGA-based prototype will depend on the requirements of the design, the power of the tools employed and the amount of effort put in by the prototyping team. Let us consider the raw capacity of different resources in an FPGA. We use a Xilinx® 6VLX760 (or just LX760 for short) as an example and will provide a capacity figure for each resource in SoC gates.

- **Logic:** FPGA implements logic in highly configurable cells combining look-up tables with optional output FFs, carry logic and other special elements for efficiently mapping logic.
- The LX760 has approximately 750,000 look-up tables and 950,000 FFs. This is sufficient to map approximately four million SoC logic gates if fully utilized, or approximately 2.5 million gates at reasonable utilization.
- **Memory :** in addition to the look-up tables, which can be configured as small memories, dedicated memory blocks are also distributed throughout the device, which can be used as single- or dual-ported R/W synchronous memory blocks. These memory blocks can be connected together and form deeper or wider memory blocks and with additional built-in logic can be used to implement specialized memories such as single- or dual-clock FIFOs. These are often configured from a library of specialist memory IP available from the vendor.
- The LX760 has 720 memory blocks which can provide approximately 26Mbits of memory in various configurations (ROM, RAM, FIFO). In addition if logic cells are used as memory then a maximum 8.3Mbits of extra memory is available, but this would reduce the amount of logic available accordingly.
- **DSP resources:** it is also common for the latest FPGAs to have dedicated DSP blocks distributed throughout the device, which include MAC

(multiply/accumulate) blocks, barrel shifting, magnitude comparators and pattern detection. In addition, DSP blocks have cascading capabilities that allow them to be connected together to form wider math functions such as DSP filters without the use of logic FPGA resources.

- The LX760 has 864 DSP blocks. These are very dense blocks and many hundreds of equivalent gates of logic can be implemented in each one. For those designs which have significant arithmetic content, good use of DSP blocks will liberate many thousands of logic elements for other purposes.
- **IO:** FPGAs' IOs can be configured in many ways to comply with a variety of standards, drive strengths, differential pairs, dynamically controlled impedances etc.
- The LX760 has 1200 usable IO pins which can be combined to produce up to 600 differential pairs. Signaling between FPGAs is often a more critical resource than the logic or memory capacity.
- **Interconnect resources:** possibly the most important resource in an FPGA is the means to interconnect between the various blocks. With the exception of some special buffers, these resources are generally not explicitly controllable by the user but rather, they are used implicitly by place & route and some advanced physical synthesis tools in order to implement a design's connectivity on the FPGA.
- **Clocking resources:** a subset of the interconnect is dedicated to implementing the design's clock. These are dedicated programmable clock generators including PLLs, and global and regional clock buffers and low-skew distribution networks.
- The LX760 has 18 very sophisticated multi-mode clock managers (MMCM) blocks providing global or regional low-skew clocks.
- **Special purpose blocks:** finally, some devices have hard-macro blocks which implement specific functions such as Ethernet MACs, PCI Express interface blocks, selected CPU cores or high-speed serial transceivers (e.g., SERDES). These macros help implement industry standard peripheral interfaces such as PCI Express or Ethernet.

Recommendation: given the dedicated nature of the special-purpose resources, SoC logic will probably not map transparently into these resources. In order to use the special purpose resource, some SoC design blocks may need to be swapped with FPGA equivalents. When such design changes are made, it should be understood that the functional behavior of the new block may not be identical to the original.

For more FPGA resources details refer back to chapter 3.

For more information on replacing SoC design blocks with equivalent (or near equivalent) FPGA cores see chapter 10 on handling IP.

4.2.5. How big is the whole SoC design in FPGA terms?

This estimate may be critical to the scale of system to be used. The largest of FPGA currently available at time of writing this manual is the Virtex®-6 LX760 from Xilinx. As we have noted above, this device has a variable mix of over 4 million SoC-equivalent logic gates, approximately 25Mbits of memory, and over 850 48-bit multiply/accumulate blocks. As we write this, that resource list seems impressive but if the history of FPGA has taught us anything it is that the FPGA which invokes awe today will be commonplace tomorrow. Therefore if this manual is being read some years after its initial publication, please scale up the figures to whichever behemoth FPGA is under consideration.

How will the FPGA's impressive resource list be able to handle the SoC design? The answer, of course, is that it is design-dependent. Given the "flip-flop rich" FPGA architecture, highly pipelined designs with a high ratio of FFs to combinatorial logic will map better into FPGA than designs with lower ratios. It will also most likely run at a faster clock rate. We do not always have the luxury of receiving an SoC design which is conveniently FPGA-shaped and so there are a number of steps which are usually required in order to make the design FPGA-ready. These are detailed in chapter 7.

The mix of the different resources varies device to device, as some devices have more of one type of resources at the expense of others. A casual use of gate-count metrics may lead to misunderstanding and the safest way to estimate the necessary FPGA resources for the design is to use FPGA synthesis upon the design to gain an estimate of total FFs, logic cells and memories. This requires the RTL to be available, of course, and also usable by the FPGA synthesis tool of choice.

It's important to note that in terms of gate-count metrics, FPGA technology and SoC technology do not compare well, and FPGA gate counts are given as a rough approximation.

Even if all SoC logic may be mapped into FPGA, the utilization may be limited owing to one or more of the following reasons:

- **The design:** some designs map more effectively into FPGA resources than other designs due to how well they fit FPGA architecture and resources. As described above, since FPGA technology is flip-flop rich, designs with a higher FF to combinatorial logic ratio are likely to achieve higher effective gate count levels than designs having a lower ratio.
- **Clock resources:** while having multiple clock regions/zones, FPGAs have a finite resource of PLL, clock multiplexers and on-chip clock routes (refer back to chapter 3). A closer look at the available clocking resources and clock zones restrictions in the selected FPGA is necessary for multi-clock designs.

- **FPGA routing resources:** FPGA's useable logic may be limited by the availability of routing resources which may vary from design to design. “Highly connected” designs may exhaust FPGA routing resources in some areas and restrict access to FPGA resources in that area, possibly rendering them unusable. In addition, such designs are likely to run at reduced clock rates as compared to “lightly-congested” designs.
- **FPGA IO pins:** it is very common with modern SoC designs for the FPGAs in a multi-FPGA partitioned design to run out of pins before they run out of logic or memory. Intelligent partitioning, which balances resources between FPGAs, may still need to use multiplexing to route all signals between the FPGAs (see chapter 8).

Recommendation: FPGA designs with high utilization levels are less likely to run at the peak clock rates. Also, highly utilized designs run the risk of exceeding available resources if or when the design grows because of mid-project design changes. While FPGA utilization levels of over 90% are possible, it is recommended to limit initial utilization levels to 50%.

4.2.6. FPGA resource estimation

In estimating the FPGA resources used in a given SoC design, the resources that need estimation are: FPGA IO, random logic, FFs, memory blocks, arithmetic elements and clocks. Manually estimating resources such as memory blocks, multipliers and high-speed IO is fairly straight forward and fairly accurate but estimating random logic and FFs is more difficult and the margin for error is greater. Therefore it is advised to first establish if the design will fit in an FPGA based on the special resources and as soon as the design is synthesizable run it through the synthesis tool for an accurate FPGA resources usage estimate. If the design is synthesizable, it is recommended to use a quick synthesis evaluation that will indicate expected resource utilization.

Once the FPGA resources utilization levels for a given design are available, we need to establish the utilization level target for the maturity level of design. As a general rule, the higher the FPGA utilization level the longer it will take to process the design (synthesis, place & route etc.) and the slower the system clock will run due to greater routing delays. In addition, during the prototyping project the design is likely to change and some diagnostics logic may be added in the future, so the number of FPGAs in the system should be considered conservatively.

On the other hand, distributing the design to many FPGAs adds cost and implementation complexity. While design size and performance can vary from design to design, it's recommended the utilization levels should initially be below

50%. Even lower utilization is recommended for partial designs that are expected to grow. It is always recommended to overestimate resources because project delays caused by underestimation can be more costly than the price of unused FPGA devices purchased based on overestimation.

The synthesis tool will then provide a resource list for the design as a whole, for example, Synplify® Premier FPGA synthesis tools provide a list such as the one shown in Figure 45 . We see the results for a large design mapping “into” a single Xilinx® 6VLX760 and even though the report may show that the resource utilization is far in excess of 100%, this is not important. Synplify Premier’s Fast Synthesis mode is particularly useful for this first-pass resource estimation.

Figure 45: Resource usage report from first-pass FPGA synthesis

```
IOIO Register bits:           426
Register bits not including IOIOs: 1877990 (198%)

RAM/ROM usage summary
Single Port Rams (RAM32X1S): 110
Dual Port Rams (RAM32X1D): 21
Simple Dual Port Rams (RAM32M): 307
Block Rams : 1339 of 720 (186%)

DSP48s: 657 of 864 (76%)

Global Clock Buffers: 6 of 32 (18%)

Mapping Summary:
Total LUTs: 1919724 (253%)
```

In addition, some thought should be given to how much of the FPGA resources it is reasonable to use. Can the devices be filled to the brim, or should some headroom be allowed for late design changes or for extra debug instrumentation? Remembering that a key aim of FPGA-based prototyping is to reduce risk in the SoC project as a whole, it follows that squeezing the last ounce of logic into the FPGAs may not be the lowest risk approach. In addition to leaving no margin for expansion, it is also true that FPGA place & route results degrade and runtimes increase greatly when devices are too full. A guideline might be to keep to less than the 75% utilization, which is typical for production FPGA designs and for prototyping projects, 60% or even 50% being not unreasonable. This will make design iteration times shorter and make it easier to meet target performance.

From the results in Figure 45 we can see that the limiting resources are the LUTs (Look-up-Tables). For this design to fit into Virtex-6 technology on our prototyping board with 50% utilization, we will need a number of FPGAs calculated as follows:

$$\begin{aligned}
 & \text{Total LUTs required / LUTs in one device} \times 50\% \\
 &= 1919724 / 758784 \times 0.5 \\
 &= \text{approximately five 6VLX760 FPGAs.}
 \end{aligned}$$

Therefore, our approximation tells us that our project will need five FPGAs.

4.2.6.1. Size estimates for some familiar SoC blocks

It may help to get a feel for the resource in an FPGA by considering the sizes given, in FPGA terms, for some familiar blocks in Table 8.

Table 8: FPGA resources taken by familiar SoC functions

IP / SoC function	FPGA resources required
ARM Cortex®-A5 (small configuration)	Two Virtex-5 LX330 at 50%
ARM Cortex®-A9	One Virtex-5 LX330 at 80%
ARM Mali™-400 w/3 pixel processors	Four Virtex-5 LX330 at 50%
ARC AS221 (audio processor)	One Virtex-5 LX330 at 10%
ARC AV 417V (video processor)	One Virtex-5 LX330 at 60%

These are only rules of thumb as each of these IP or processor examples have a number of configurations, memory sizes and so forth. However, we can see that even as FPGAs continue to get bigger, the first law of prototyping still holds true.

Recommendation: To manually estimate the required FPGA resources can be difficult, time consuming and may be quite inaccurate. It's often more effective to run preliminary synthesis and quickly find out the FPGA resource usage levels and have initial performance estimation.

4.2.7. How fast will the prototype run?

Performance estimation is tightly coupled to the resource utilization levels and the FPGA performance parameters. Similar to FPGA resource estimation, performance

estimation for the special blocks is easily extracted from the timing information in FPGA datasheets. Estimating the overall performance is much harder. However, a fairly good performance estimation of the whole design is easy to obtain from the synthesis tool. Although the synthesis tool takes in account routing delays, the actual timing depends on the FPGA place & route process and it may differ from the synthesis tool's estimation. While the difference can be significant with higher utilization levels as routing can become more difficult, the initial timing estimation is quite a useful guide.

FPGA performance greatly depends on the constraints provided to the synthesis tool. These constraints direct the synthesis tool and subsequently the place & route tools how to best accomplish the desired performance. For more details on how constraints affect the FPGA implementation, refer to chapter 7.

There are a number of factors affecting the clock rate of a design mapped into a multi-FPGA system, as described below:

- **Design type:** highly pipelined designs that map well into FPGA architecture and leverage their abundant FFs are likely to run faster than designs that are less pipelined.
- **Design internal connectivity:** designs with complex connectivity, where many nodes have high fan-out, will run slower than designs with lower fan-out connectivity due to the likely longer routing delays. It will also be harder to find places in the design where a partitioning solution is possible with few enough IO to fit within an FPGA footprint. Higher interconnected designs will more likely require multiplexing of multiple signals onto the same IO pins.
- **Resource utilization levels:** typically the higher the utilization levels, the more congested the design, resulting in longer internal delays and a slower clock rate. Below, we explore the recommendations for device utilization.
- **FPGA performance:** the raw performance of the FPGA itself. Even with the most tailored and optimized design, we will eventually hit an upper limit for the FPGA fabric. In most cases, however, the non-optimized nature of the design and the efficiency of the tool flow will be seen as a limit long before the absolute internal clock rates of the FPGA.
- **Inter-FPGA timing:** in a multi-FPGA system, FPGA-to-FPGA clock skews and connectivity delays can limit the system clock rate. While FPGAs can theoretically run internal logic at clock rates of hundreds of megahertz, their standard IO speed is significantly slower and is often the prevailing factor limiting the system clock rates.
- **External interfaces:** as we shall explain later, SoC designs mapped into prototyping systems are likely to run at a slower clock rate than the SoC's target clock. Other than the expected performance penalty, this is not a big

issue for a closed system running with no external stimuli, or a system for which the stimuli can run at a slower rate to match the system's clock rate. There may well be situations where the prototyping system must interface with stimuli that cannot be slowed down.

- **Inter-FPGA connectivity:** when all inter-FPGA connectivity is exhausted, pin multiplexing can be used. In time-domain multiplexing a number of signals share a single pin by running at a faster clock than the data rate of the individual signals multiplexed together. For example, when multiplexing four signals each running at the rate of 20MHz the combined signal will need to run at least at the rate of 80MHz and actually more, in order to allow for timing for first and last signals sampled. Since FPGA-to-FPGA data rate is limited by the physical FPGA pins and the board propagation delays, the effective data rate of the individual signals in this example will only be less than a quarter of the maximum inter-FPGA data rate.

Recommendation: pin multiplexing can severely limit systems' clock, so it's critical to evaluate the effect of partitioning on inter-FPGA connectivity during the early feasibility stage of the prototyping project. More detail on pin multiplexing is given in chapter 8.

4.3. How many FPGAs can be used in one prototype?

When designs are too large to fit into a single FPGA and still remain within recommended utilization levels, we must partition the SoC design over multiple FPGAs. While there is no theoretical limit to the number of FPGAs in a system – and some designs will map well into large multiple FPGA systems – the eventual limit on the number of FPGAs in a prototyping system will depend upon both the design itself and the limitations of the prototyping platform.

In general, the following points will limit the number of FPGAs in a system:

- **FPGA-to-FPGA connectivity:** as designs are split into more FPGAs, inter-FPGA connectivity typically grows, and depending on the design and how it's partitioned, it may exceed the available connectivity in a given system. Inter-FPGA connectivity is bounded by the available inter-FPGA connectivity in a given system. Depending on the systems, the inter-FPGA connectivity may be either fixed or programmable to some extent. A common technique to overcome inter-FPGA connectivity bottlenecks is to use high-speed pin multiplexing schemes in which multiple signals "time-share" a single connection. The time-domain pin multiplexing, however, requires a high-speed clock which may limit the system clock rate due to the timing limitation of the physical connection between the FPGAs.

- **Signal propagation:** since propagation delays to and from an FPGA's IO pads are typically longer than propagation delays within the FPGA, signal propagation among FPGAs is typically the timing critical path and directly affects the system clock rate. Excessive FPGA-to-FPGA delays on the board (including long signal-settling times) will reduce timing margins and may limit the system's clock rate. Signal propagation issues are more significant with greater number of FPGAs in the system due to the physical implementation, especially when connecting multiple boards together where signals go through multiple connectors and connection media (cables, other boards) and ground returns and reference may become marginal.
- **Clock distribution:** proper clock distribution in a synchronous multi-FPGA system is critical to its proper operation. Specifically, the clocks driving signals out from one FPGA and the clocks used to clock-in signals from other FPGAs must have minimal skew between the FPGAs exchanging data as to not violate setup and hold times. As systems grow larger with more FPGAs, the physical clock distribution may become harder to implement with an acceptable skew especially in scalable systems where multiple boards are connected together.
- **Manual design partitioning:** as the number of FPGAs in a system grows, partitioning becomes increasingly more complex, and manual partitioning may be impractical altogether. This may prove especially difficult if the partitioning needs to be modified often as the design changes.
- **Managing multiple FPGAs:** while not a technical barrier, the more FPGAs there are in a system, the more cumbersome the overall process is requiring a greater management effort. Specifically, a number of FPGAs may need to be re-processed (synthesis, place & route) with each design iteration, and processing multiple FPGAs in parallel requires multiple tool licenses for the software tools, otherwise the process becomes serial, taking longer to complete. In addition, each FPGA needs to be managed in terms of pin assignments, timing constraints, implementation files, revision control etc., which adds to the overall project engineering administration overhead.

When considering large multi-FPGA systems, it's important to evaluate how well they address the above issues, and how well they scale. When either making boards in-house or buying in ready-made systems, such as Synopsys' HAPS® and CHIPit®, the same FPGAs may be used but it is the way that these can be combined and interconnected that may be the limiting factor as projects grow or new projects are considered. The next three chapters aim to cover the ways that platforms can be built and configured to meet the needs of the specific project and/or subsequent projects.

Recommendation: early discovery of the design's mapping and implementation issues is critical to the effectiveness of the prototyping effort. Using a partitioning tool such as Synopsys' Certify® can simplify and speed up the partitioning process.

4.4. Estimating required resources

The various implementation and debug tools were described in chapter 3 but as a recap, a number of software tools are needed to map and implement the SoC into FPGAs. All tools come with their respective design environment or GUI, but can also be invoked from a command line or a script so that the whole implementation sequence can then be more efficiently run without supervision, for example, overnight.

The time taken to run the design through the tools will vary according to a number of factors including complexity, size and performance targets as we shall see below but runtime will always benefit from using more capable workstations and multiple tools in parallel. Readers may have some experience of FPGA design from earlier years, where a PC running Microsoft Windows® would suffice for any project. Today, however, our EDA tools used for prototyping are almost exclusively run on Linux-based workstations with more than 16Gbytes of RAM available for each process. This is mostly driven by the size of designs and blocks being processed and in particular, FPGA place & route is generally run on the whole design in one pass. With today's FPGA databases having some millions of instances, this becomes a large processing exercise for a serious workstation.

In addition, we can increase our productivity by running multiple copies of a tool in parallel, running on separate workstation processors. For example, the synthesis and place & route on the different FPGAs after partitioning could be run in parallel so that the total runtime of the implementation would be governed only by that FPGA with the longest runtime.

4.5. How long will it take to process the design?

As a general rule, the higher the utilization level the lower the achievable clock rate and the longer the FPGA processing time. As we saw in chapter 3, the implementation process consists of the front-end and the back-end. Between them they perform five main tasks of logic synthesis, logic mapping, block placement, interconnect routing, finally, generation of the FPGA bit-stream. These tasks are summarized in Table 9.

Depending upon the tool flow, these tasks will be performed by different tools, the proportion of the runtime for each step will vary. For example, the traditional back-end dominated flow has most of these tasks performed by an FPGA vendor's

dedicated place & route tools, hence the runtime is mostly spent in the placement and routing phases.

Table 9: Front-end and Back-end tasks for each FPGA

Task		Description
Front-end	Logic synthesis	RTL to behavior/gates
	Mapping	Gates to FPGA resources
Back-end	Placement	Allocate specific resources
	Routing	Connect resources
Bitstream generation		Final FPGA “programming”

However, it is also possible to employ physical synthesis for the FPGA in which the front-end also performs the majority of the placement task. Consequently, more of the overall runtime would be spent in the front end.

Table 10: Tool runtime as proportion of whole flow (typical average)

Task	Proportion of runtime Flow 1		Proportion of runtime Flow 2	
Logic synthesis	Front-end	10%	Front-end “physical synthesis”	10%
	Back-end	20%		35%
Placement		30%		35%
		35%	Back-end	15%
Routing		5%		5%
Bitstream generation				

Table 10 shows the physical and non-physical flows side-by-side and gives the proportion of time typically spent in each step. These are only typical numbers of the relative time taken for each part of the process; real time will depend on the device utilization levels and timing requirements.

As we can see, placement and routing generally take longer than synthesis and as designs become larger and timing constraints more demanding, this becomes even more apparent. For example, a large FPGA at 90% utilization might easily take 24 hours or more to complete the whole flow; three quarters of that time being spent in place & route. When prototyping, this long runtime may be a large penalty to pay just to avoid using another, admittedly expensive, FPGA device. It may be better in the long run to use three FPGAs at 50% utilization, rather than two FPGAs at 75% each.

In general, FPGA physical synthesis is used for FPGA designs that are intended for mainstream production and require the best possible results. For FPGA-based prototyping, physical synthesis might be focused on one particular FPGA on the board which needs more help to reach the target performance.

Recommendation: FPGA designs with high utilization levels will take longer to process so while FPGA utilization levels of over 90% are possible, it is recommended to limit initial utilization levels to 50% in order to allow faster turn-around time on design iterations.

4.5.1. Really, how long will it take to process the design?

Avoiding answers that involve pieces of string, we should expect runtime in hours and not days. The aim should be a turn-around time such that we can make significant design changes and see the results running on the board within a day. Indeed, many prototyping teams settle into a routine of making changes during the day and starting a re-build script to run overnight and see new results the next morning. Automating such a script and building in traps and notifications will greatly help such an approach.

Long runtime is not fatal as long as we are careful and we get a good result at the end. Runtime becomes a problem when careless mistakes render our results useless, for example pin locations omitted or we iterate for only small changes.

If runtimes are too long to allow such a turn-around time, then we recommend taking some steps to reduce runtime. These might include some of the following:

- **Add more workstations and licenses:** this allows greater parallel processing and balancing of runtime tasks.
- **Lower FPGA utilization:** repartition design into more FPGAs. This may take some time but it may be an investment worth making. Total runtime can vary widely as a function of utilization level of the device.
- Note: It may be preferable to process six devices at 50% each rather than four devices at 75% each.
- **Relax timing constraints:** on less critical parts of the design it is possible to lower the timing goals to reduce tool runtime. Place & route runtime depends not only on utilization and other factors can have an even greater effect, including applied constraints, number of global clocks, and the number of clocks driving BlockRAMs. Basically, the more complex the task given to place & route, the longer the runtime.
- **Use incremental flows:** both synthesis and place & route have built-in incremental flows which reduce runtime by not re-processing parts of the design that are unchanged.

- **Use quick-pass flows:** some tools have options for turning off some optimization steps at the cost of lower quality of results. For example, the Fast Synthesis option in Synopsys FPGA synthesis tools.
- **Relax timing model:** in extreme cases, the timing model for the FPGA can be relaxed on the assumption that the device will not be used outside the lab and therefore temperature of voltage extremes built into the timing model will not apply.

Recommendation: To speed the implementation cycles, it is recommended to relax the timing constraints where possible as to minimize implementation times. Once the initial issues are resolved, and the implementation cycles are not as often, the more strict timing constraints can be applied.

4.5.2. A note on partitioning runtime

Somewhere along the way, it is probably going to be necessary to partition the design into multiple FPGAs. As we saw in chapter 3, partitioning could occur before logic synthesis i.e., by manually splitting the design into multiple FPGA projects, or after logic synthesis, by some database manipulation prior to mapping.

The runtime of partitioning tools is a relatively small part of the runtime for the whole flow and is not as greatly impacted by increased design capacity or constraint, as it is by the IO limitations. However, the automation of a partitioning flow may impact the success or otherwise of a scripted flow. We need a robust and tolerant methodology in the partitioning tool that can cope with changes in the design and, for example, not terminate when a new gate has appeared in the design because of a recent RTL change. The partitioner should be able to assume that if the new gate's source and destination are in a given FPGA, then that gate itself should be automatically partitioned into that same FPGA. It would be rather frustrating to arrive at the lab in the morning after starting an overnight build script to discover a message on our screen asking where we want to put gate XYZ. So a partitioner's runtime is less important than its automation and flexibility.

4.6. How much work will it be?

Implementation effort is divided here into the initial implementation effort, which are mostly performed only once in the prototyping project, and the subsequent implementation efforts which are repeated in each design iteration or “turn.”

4.6.1. Initial implementation effort

Initial activities for setting up the prototype mostly take place the first time the design is implemented where the infrastructure and implementation process are created and are then reused in subsequent design turns. The initial implementation efforts include the following:

- **SoC design modifications:** modifications done to the SoC design to better map it into FPGAs. This typically includes design optimization for performance (pipelining), trimming parts of the design not needed for prototyping, and replacing SoC structures with available FPGA structures, such as clocking circuits, memory structures etc.
- **Partitioning:** the process of dividing the SoC design into multiple FPGAs per the platform of choice.
- **Design constraints generation:** the process in which FPGA design implementation constraints are created. Typical constraints are pin placement, timing constraints and inserting synthesis directives.
- **Debugging features:** the process of configuring debugging features that will go into the design. This is an optional effort but is commonly used and typically repeated in subsequent design turns as needed.
- **Physical design implementation:** a series of processes necessary to implement the design. They include synthesis, mapping, place & route, timing analysis and bitmap generation. These activities are typically combined in implementation scripts that can be used in subsequent design turns.

The time it takes to accomplish the initial prototyping effort is usually the most significant in prototyping projects, especially for the first-time prototyper. This effort may involve learning new technologies and tools and acquiring platform and components, so overall effort may vary on a case-by-case basis. There is also the significant effort of adapting the SoC design for use in the FPGA, which is obviously design dependent.

Recommendation: To set expectations correctly, the initial implementation effort (excluding the platform and tool evaluation and training), for a four-FPGA system with each device resource utilization around 50% and a relaxed clock rate, might take ten to 16 engineer-weeks of effort, which is five to eight weeks for a typical two-engineer team.

4.6.2. Subsequent implementation effort

After the design has been successfully implemented once in the FPGA system and only small design modifications are needed, such as a result of bug discoveries, we will need to make design changes. Typically these will only have a small impact on partitioning and design constraining and therefore mostly involve the design implementation phase once the whole implementation process is set up. Assuming the design implementation is scripted into “build” or a “make” files, this effort is fairly small and is limited to implementing the design changes, running the implementation scripts and reviewing various report files.

Recommendation: where possible, use the incremental design features through the implementation chain to minimize implementation time for minor design changes.

The time for subsequent implementation depends on the extent of design modification compared to the previous run:

- **Minor modifications:** for minor changes, such as changing debug probes or small logic changes to the design itself, re-implementation of the design requires only synthesis, place & route. Implementation tools can be configured to operate in an incremental mode where only blocks that have changed require re-synthesis and place & route, which translate into significant processing time savings. For such cases implementation time can typically be one to four hours per FPGA.
- **Moderate modifications:** at this point the design may have grown beyond the initial 50% utilization levels target, and the timing requirements may be harder to accomplish. If the design changes are significant then implementation may involve partition changes. Therefore, such iteration may take as much as a couple of days.

4.6.3. A note on engineering resources

First-time prototypers should not make the mistake of thinking that making an FPGA-based prototype is a summer job for the intern. For a successful and productive prototyping project there needs to be a good understanding of the design to be prototyped, of the FPGA technology, of the tools necessary and prototyping optimization techniques. Since prototyping is often critical to the overall SoC project schedule, it is recommended to dedicate at least one engineer with a good RTL knowledge, a good understanding of FPGA technology and tools, and good hardware and at-the-bench debugging skills. These are not skills typically found in SoC verification teams and so some consideration of building and maintaining such expertise in-house should be undertaken.

Larger corporations with multiple prototyping projects typically retain a team of engineers specifically for FPGA-based prototyping tasks. Such a valuable expert resource has a high return-on-investment, as SoCs, and the software running on them, benefit from better verification.

In some cases it is possible to share some of the tasks among multiple engineers. For example, to identify multi-cycle paths to relax the timing constraints or to make RTL changes to improve timing, the SoC engineers may be a good choice to implement these tasks since they are most familiar with the design and can resolve these issues more effectively.

Recommendation: for a first-time prototyping project, it is recommended to make use of consultants who can provide the necessary front-to-back prototyping expertise. Employing and learning from such experts is a very productive way to get up to speed with prototyping in general while also benefiting the current project.

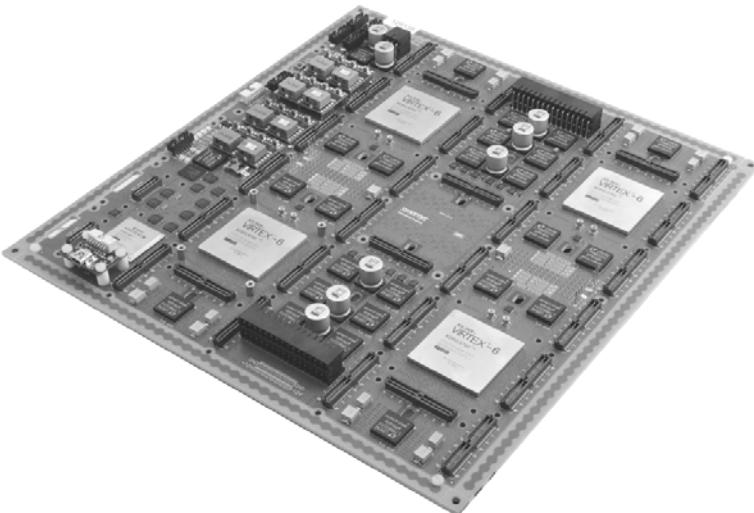
4.7. FPGA platform

After estimating the logic resources needed, the size of the system should become evident. Specifically, the type of FPGAs to use, the number of FPGAs needed, the interconnect requirements, and additional hardware such as external memories, controllers, analog circuitry etc., that will be added to the FPGA system. At this point in time, the decision whether to buy an off-the-shelf FPGA system or to develop it in-house must be made. Some of the important considerations are how well the system scales with design size changes and the potential reuse of the system for future prototyping projects, where the more “generic” the platform is, the more re-usable it will be in future projects.

As an example of the types of ready-made FPGA platforms available, Figure 46 shows a platform from a Synopsys HAPS® (High-Performance ASIC Prototyping System) family of boards.

This example shows a HAPS board which has four Virtex-6 LX760 FPGAs, plus a smaller fifth FPGA, which is used to configure the board and also holds the supervisor and communications functions. There are other HAPS boards which have one or two LX760 devices each but with the same level of infrastructure as the board with four FPGAs. So we have the same resource in effect in three configurations i.e., single, double and quadruple FPGA modules which can operate independently or be combined into larger systems. The HAPS approach also includes different connectivity options and separate add-on boards for various hard IP, memory, and IO options.

Figure 46: Example of a ready-made FPGA platform, HAPS®-64



The importance of such flexibility becomes apparent when platforms are to be reused across multiple prototyping projects. The options and the trade-offs between making project-specific boards and obtaining third-party ready-made boards like HAPS are described in more detail in the “Which platform?” chapters (5 and 6) so we shall not cover in any more detail here.

4.8. Summary

We have now covered all the reasons why we might use FPGA-based prototyping and which types, device and boards might be most applicable in given situations. We have decided which parts of the design would most suitable for prototyping and how we can get started on our project.

The most important point to be re-iterated from this chapter is that there is an effort ‘vs’ reward balance to be considered in every project. With enough time, effort and ingenuity, any SoC design can be made to work to some degree in an FPGA-based prototype. However, if we start with realistic and achievable expectations for device usage and speed, and omit those design elements which bring little benefit to the prototype but would take a great deal of effort (e.g., BIST) then we can increase the usefulness of the prototype to most end-users.

It is important to explore the different options for target FPGA platforms at an early stage. The most obvious choice from the start is whether to design and make your own specific boards or to obtain boards from an external supplier. The next two chapters will discuss technical aspects of the platform choice, starting with this chapter, in which we will focus on making your own boards in-house. There is also some detail in Appendix B on making an informed comparison from an economical and business perspective.

5.1. What is the best shape for the platform?

In earlier chapters we explored the many possible aims for an FPGA-based prototyping project; by the time we get to choose the platform, the basic needs for the project will probably be well understood. On the other hand, there may be a whole series of projects for which a platform is intended, requiring a more forward-looking and generic solution to multiple needs. The platform's topology, that is, the physical and electrical arrangement of its resources will need to meet those needs.

Let us consider the topology needs as divided into the following areas:

- Size and form factor
- Modularity
- Interconnect
- Flexibility

each of which we will now give further consideration.

5.1.1. Size and form factor

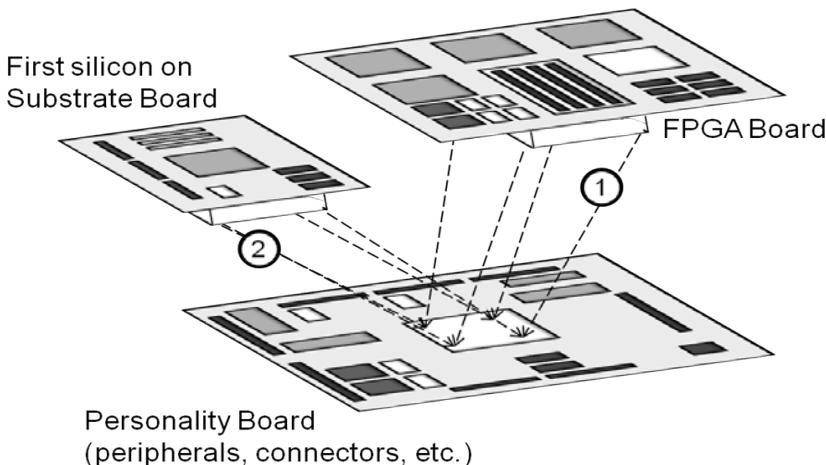
Clearly, an FPGA-based prototype will take more space in the final product than that occupied by the intended SoC, but in most cases we are not looking for an exact physical replacement of one by the other. How close to final form-factor does the FPGA-based prototype need to be?

In many cases, umbilical cable may be used to connect the FPGA platform to the in-system socket of the intended SoC. In that case, electrical and timing considerations would need to be understood. For example, do the FPGA IO pins have the electrical characteristics required to drive the cable load as well as the final SoC load. Would the cable require high-drive buffers to make this possible? If so, we should also then account for the buffer delays as part of the IO constraints for the implementation tool flow.

The advantage of having an umbilical cable to a final SoC socket is that the platform may be more easily used as a demonstration model for external suppliers, partners or investors. If this is an important aim then it might also be worth considering the use of a complimentary virtual prototype for early demonstrations, potentially linking this to part of the FPGA platform in a combined hybrid prototype environment.

It is not common to find these umbilical topologies in real-life FPGA-based prototyping projects. In fact, a more widespread approach is almost the opposite with the SoC first sample silicon being more often plugged into a pre-prepared location on the FPGA-based prototype platform, rather than the other way round as discussed above. This requires a hardware topology in which there is a physical partition at the SoC pins; that is, between the contents of the SoC (i.e., pins-in) and the rest of the system (i.e., pins-out). That may seem natural and obvious but in-house boards can often mix everything together, especially when the boards are intended to be used only for a single project.

Figure 47: FPGA-based Prototyping platform employing a personality board



A diagram of this pins-in/pins-out hardware topology is shown in Figure 47. Here we see a personality board for the pins-out portion of the prototype, including peripherals which represent the interface for the SoC with its final target product. The personality board is connected via a header to the pins-in portion of the design which is modeled on an FPGA board. Upon receipt, the first silicon is mounted on a substrate board and fitted onto the header in place of the FPGA board.

User often tell us that when first-silicon is introduced into a previously working prototype platform in this way then software can be up and running in only a matter of hours. This is an excellent example of how some forethought regarding the topology of the FPGA-based prototyping platform can greatly improve productivity later in the project.

5.1.2. Modularity

The specific case of a pins-in/pins-out topology partition leads into a more general discussion of modularity. When building an FPGA-based Prototyping platform, should all components be placed onto one board to potentially increase FPGA-to-FPGA performance or should different types of component (e.g., FPGAs, peripherals, memories) be placed on separate boards, to potentially increase flexibility? Or is there perhaps some hybrid arrangement of the two? There is no one right way for all cases but in general, what starts as a question of flexibility versus performance, becomes more a matter of cost, yield and manufacture, as we shall discuss later.

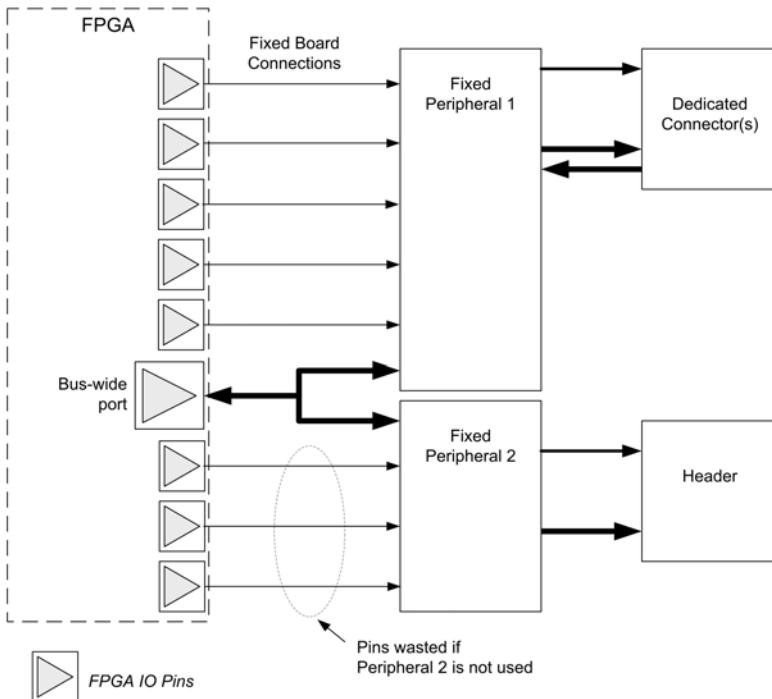
Considering just the flexibility versus performance question for the moment, an arrangement of mother-board and daughter-boards may be a good compromise. If the boards are intended for multiple projects, it makes sense to group components into those which will always be required for every project and those which are only required occasionally. Ever-present components may then be placed on a motherboard along with the FPGAs and other global resources. Those components which are specific to a given project may then be added as daughter boards as required. For follow-on projects, the hardware re-design will then be limited to one or two daughter boards rather than an entire platform.

5.1.2.1. Mother and daughter cards vs. “all-on-board”

It is often tempting to make one large single board to fit the whole design and indeed, this can be good for a very targeted or application-specific solution, especially if it does not require a large number of FPGAs and has a well defined set of peripherals. For example, if we are always expecting to prototype video graphics designs, then we might always add an on-board VGA or other screen driver port. If

we are in a networking design team, then an Ethernet/IP port would probably always be necessary.

Figure 48: Inefficient use of FPGA IO pins owing to dedicated peripherals



However, as a note of caution, peripherals on an FPGA motherboard are likely to be connected directly to FPGA pins, as shown in Figure 48.

If the peripheral is not used for a particular design, then the FPGA IO pins would still be connected to the unused peripheral and therefore not available for any other purpose, limiting the overall capacity of the motherboard. In addition, effort must be made to ensure that the unused peripheral remains dormant or at least does not interrupt the operation of the rest of the board.

The common solution to avoid this compromise is to place the peripheral on a separate daughter card so that it can be connected into the system when necessary and otherwise removed. If we have an attitude to on-board peripherals of "if in doubt, put it on" then we may severely limit the most valuable resource on the motherboard, i.e., the IO pins of the FPGAs themselves.

5.1.3. Interconnect

As mentioned, the FPGA IO pins and the connections between the FPGAs should be considered the most valuable resources on the platform. This is because they are the most likely to be 100% used, especially when a non-optimal partition is employed. What is the best Interconnect topology for a given design and partition (or for a given set of designs, assuming reuse is desirable)?

Given that in-house platforms will often be designed and built with a specific SoC project in mind, the connectivity between the FPGAs will tend to resemble the connectivity between major blocks at the top-level of the SoC design. This is especially likely if the SoC has been designed with the platform in mind. This may seem a circular argument but it is not. This is because many designs are derivative of previous versions and also because Design-for-Prototyping methods will tend to group the SoC's top-level hierarchy into FPGA-sized blocks. With the platform interconnect being fixed and matching the SoC top-level connectivity the partitioning of the design becomes easier. The downside is that the reuse of the board for follow-on projects becomes harder, as will be seen in the next section.

For a platform to be more generic and therefore more likely to be re-usable across multiple projects, the interconnect resource should be as design-independent as possible, although perhaps constrained by some ever-present design elements such as the width and number of buses in the SoC or a particular data-path arrangement between the SoC and system-level elements. If projects are always standardized around a specific bus specification, for example, then inter-FPGA connection needs can be anticipated to some degree.

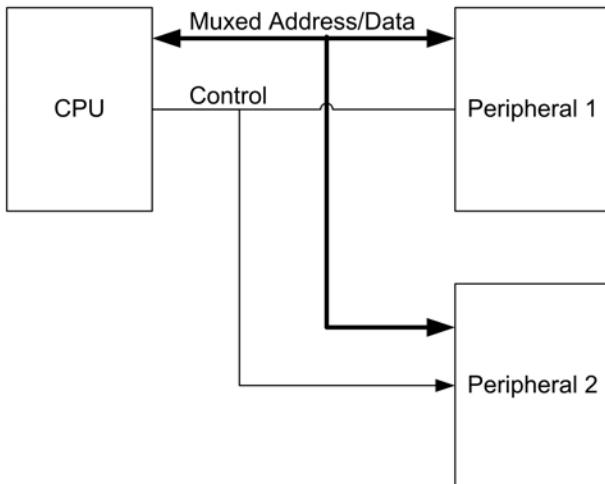
For example, if a 64-bit AHB™ bus is always employed then connectivity on the board can be provided to carry the bus between all FPGAs. It should always be remembered, of course, that an AHB bus and other standardized bus systems require many more pins than may be inferred by their nominal width. For example, a 64-bit bus might have 64 bits of address, 64 bits of write data, 64 bits of read data and 10 or 20 or more control and status signals, e.g., rd, ale, rdy etc. This would total over 200 or more connections at each FPGA that resides on the bus, which is a sizeable proportion of the total inter-FPGA IO connections available. Even if the address and data were multiplexed, the number of global pins and traces would be over 130.

Consider the example in Figure 49, which shows a simple 64-bit multiplexed address-data bus connection between a CPU and two peripherals.

It is quite possible that a situation would arise where this circuit were mapped to three FPGAs as shown in Figure 49 with a fourth FPGA on the same board not required to house a bus peripheral, but instead some other part of the design.

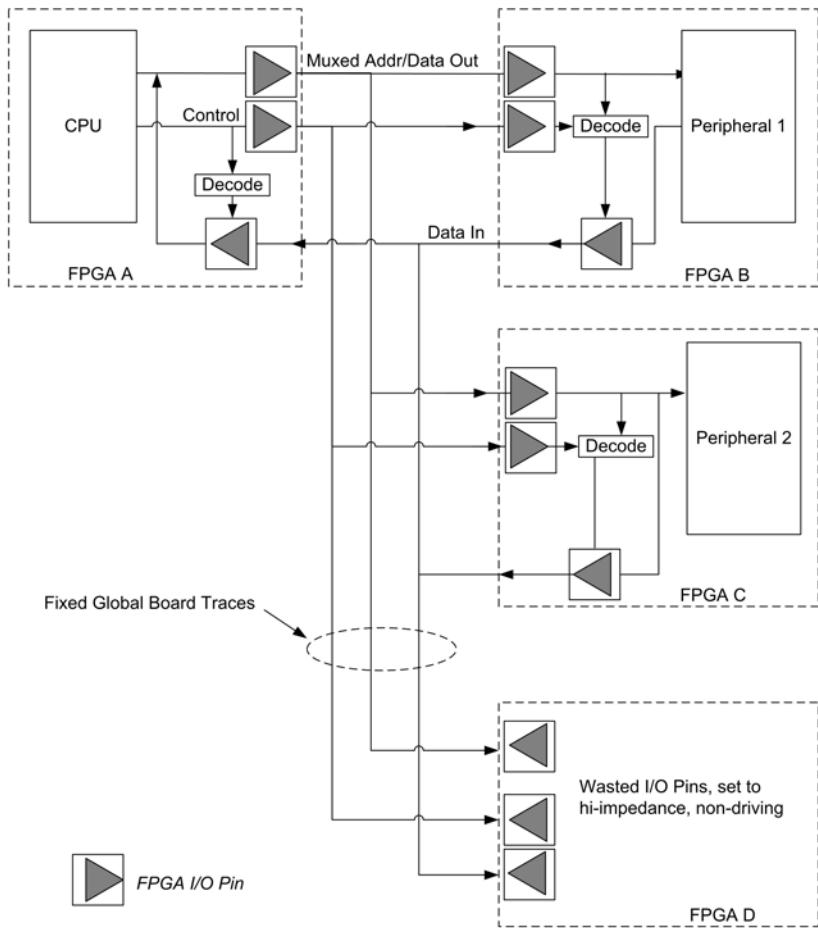
Without care and foresight, wider buses and datapaths can run out of interconnections on a given board. It is necessary to fully understand the top-level topology and interconnections at the SoC top-level but it is also recommended to build flexibility into the board interconnect resources. This could be done using cables or switch matrices rather than using fixed traces in the PCB itself.

Figure 49 : Multiplexed address-data bus before partitioning (compare with Figure 50)



If the board has fixed global resources routing the bus to every FPGA then the fourth FPGA (FPGA D in the example shown in Figure 50) would not be able to use 130+ of its precious IO pins. These would need to be set to high-impedance, non-driving mode so as not to drive or excessively load the traces being used to route the bus between the remaining FPGAs. We shall revisit this example in chapter 6 when we consider commercially sourced boards and how the limitations of interconnect can become a very critical factor.

Figure 50: Multiplexed address-data bus from Figure 49 after partitioning



5.1.4. Flexibility

How flexible does the platform need to be? It may be decided from the start of the project that the platform will only ever be used for one prototype; follow-on reuse would be a bonus but is not necessary to justify the project or its budget. This

simplifies matters considerably and is the most common compromise when designing in-house platforms. Indeed, in the authors' experience, most in-house platforms are never reused for follow-on projects. Not only because that was never the intent (as above) but also because, even when intended for reuse, the boards themselves do not have the resource or interconnect to allow it. In those cases where in-house board development was only funded by management based on a multi-project return-on-investment, if this turns out not to be possible then we might jeopardize use of FPGA-based prototyping in future projects. We can see then that building in flexibility can be highly beneficial, despite the necessary extra complexity.

For maximum re-usability, flexibility is needed in the following areas:

- Total FPGA resources
- Peripheral choice
- Interconnect
- On-board clock resources
- Voltage domains

The first three points have been considered already in this chapter so let us consider clocking and voltage flexibility. Clock network needs are very different for SoC and FPGA designs and manipulation and compromise of the SoC clocks may be required in order to make an FPGA-based prototype work in the lab (see chapter 7). Our aim should be to use platforms and tools which minimize or eradicate these manipulations and compromises.

5.2. Testability

In chapter 11 we shall explore the process of bringing up our boards in the lab and applying the SoC design to the board for the first time. At that point, we shall rely on having confidence that the board itself is working to spec before we introduce the design. The task of the board designer is not only to provide ways to test the functionality of the boards during manufacture and initial test, but also to provide extra visibility to enable the board functionality to be tested once deployed in the lab. This is a different subject to the use of instrumentation to add visibility inside the FPGAs to the design itself.

Testability is of course a consideration for in-house platform development. The ability to test all interconnect, especially for modular systems with many connectors, is essential. There are third-party tools that allow you to develop continuity suites to confirm that connectivity is complete. The ability to test continuity when platforms go down by users more than pays for itself. This also helps to diagnose blown FPGA pins during operation.

5.3. On-board clock resources

Using a modern FPGA fabric as an example, we see a remarkable variety of clock resources inside the device as shown in chapter 3. If the SoC design is small enough to be placed inside a single FPGA then as long as the FPGA has the required number of clock domains then the SoC clock network will probably be accommodated by the sophisticated clocking networks in the FPGA, including PLL, clock scaling and synchronization between local clocks and global clocks.

Having claimed that, if SoC clock networks are small enough they can be accommodated by a single FPGA, once the SoC design must be partitioned across multiple FPGAs then an extra layer of complexity is involved. In effect, the top-layer of the clock network hierarchy needs to be accommodated by the board, rather than inside the FPGA devices. One useful approach is to consider the board as a “Super-FPGA” made from the individual FPGAs plus top-level resources joining them together into the overall platform.

To replicate an FPGA’s full hierarchy of clock functionality at board level would require similar resources to those found inside each FPGA. So, for example, we would need board-level PLLs, clock multiplexing, clock dividers and multipliers, clock synchronization circuits and so forth.

In many cases, we can simplify this requirement by using tools to simplify the SoC clocking into something that we can handle on our boards. For example, converting some of the SoC design’s gated clocks into equivalent global FPGA clocks, plus relevant clock enables, in order to reduce the total number of clock domains. This gated-clock conversion can be automated without altering the RTL as will be shown in detail in chapter 7.

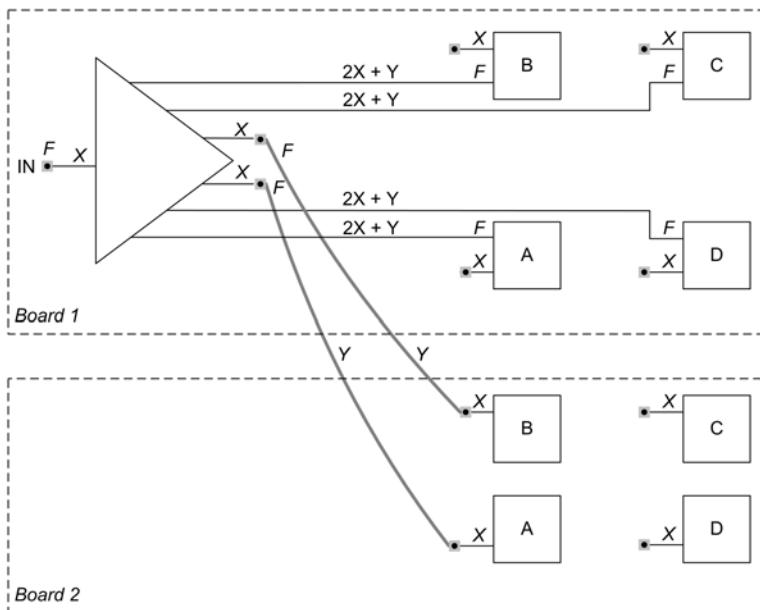
Building fully flexible clock resources into a board to meet the requirements of a variety of SoC designs requires a great deal of experience and expertise and so it may be tempting to create our board for a single project only. The board would be made with only those top-level clock and other resources necessary to map the specific SoC design. This is certainly much easier than planning for a fully flexible arrangement for all possible future FPGA-based prototypes. However, a board with inadequate flexibility in the clock resources will place greater constraint on the partition decisions should the design change to some degree during the project. It will also heavily reduce the ability for the board to be used across multiple projects because the top-level clock resources for one SoC project may not closely match those for subsequent projects. In the authors’ experience, it is limitation of the top-level clock resources that is a common reason that in-house boards are not often reusable across many projects.

5.3.1. Matching clock delays on and off board

A clocked sub-system in the SoC will be implemented in silicon with tight control and analysis of clock distribution delays. When such a sub-system is split across multiple FPGAs, we must not introduce skew into our clock networks. The partitioning of an SoC design into multiple FPGAs will require control of inter-FPGA delays and especially inter-board delays, if multiple boards are to be used. Some of this can be done by properly constraining the FPGA tools and the task might be made easier by reducing the overall target clock speed for the prototype. However, regardless of the clock speed the best way to ease this task is to design our boards with matched and predictable clock distribution resources.

As an example of delay matching, we look no further than the Synopsys® HAPS® family of boards. These are laid out with quanta of delay which are repeated at key points in clock distribution paths. In the example in Figure 51, we see two quanta of

Figure 51: Delay matching on and off board



delay, X and Y, which are used as fundamental values during the board design. Y is the delay of typical clocks along a particular standard length co-axial cable with high-quality shielding and co-axial connectors which is mass produced and used widely in HAPS-based platforms. Having a single type and length of cable with relatively constant delay meant that the on-board traces can be designed to the same

value Y. It is a small matter for PCB designers to delay match traces on the board, although the zig-zag trace routing employed may increase the need for extra board routing layers. The other quantum of delay, X, is the stub delay to and from FPGA pins and nearby connectors and is relatively simple to keep constant. The values for X and Y will be characteristic for a given family of boards, for example, for one family of boards, $X = 0.44\text{ns}$ and $Y = 1.45\text{ns}$. With care, the values of X and Y can be maintained across new board designs in order to allow easier mixing of legacy and new boards in the same prototype.

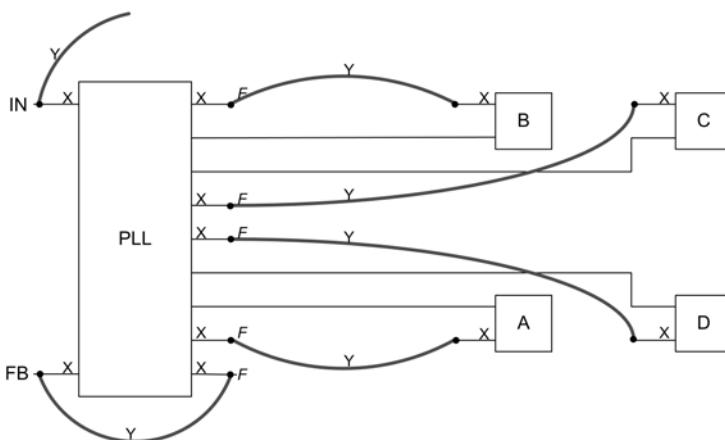
We can now see from the diagram that clock delay between the clock source and any of the FPGA clock pins will be the same value, $2X+Y$, i.e., 2.33ns for our HAPS example, and thus we have minimized clock skew across the partition.

Note, however, that the received FPGA clocks, although mutually synchronized, will not be in sync with the master clock input, F. If we wish to use F and its distributed derivatives, then we will need to employ phase-locked loops, which we will now discuss.

5.3.2. Phase-locked loops (PLL)

PLLs are extremely useful for removing insertion delay in cross-board clock

Figure 52: PLL used to re-align FPGA clocks



networks, for example, to reduce clock skew between different parts of the prototype. Figure 52 shows a board-level PLL driving four FPGAs. As above, with the use of trace-length matching, the stubs at each FPGA clock input and the stubs at the PLL outputs can be made equal to the value X. Over many board

manufacturing runs, the absolute value of X may vary but will be consistent enough for any given board.

We see also that the FPGAs are linked to the PLL outputs by equal length cables or delay Y. Co-axial cables and high-fidelity clock connections are highly recommended for best reliability and performance. We can see that the on-board delays from PLL to each FPGA are matched.

For providing the necessary clock resources at the top-level of the “Super-FPGA,” the board should include PLL capabilities, which are useful for a number of tasks. Although the FPGAs used may include their own PLL features, the boards should include discrete PLL devices such as those available commercially from numerous vendors.

5.3.3. System clock generation

The clock generation and its flexibility are critical to the reliable operation of a multi-FPGA Prototyping system. Some of these capabilities must be implemented at the board level, but others must make use of the available clock resources within the FPGAs. It is therefore important that from the beginning we understand the available clocking resources on the chosen FPGA. As mentioned in chapter 3, FPGA devices today have very capable clocking capabilities including DLLs, PLLs, clock multiplexers and numerous global and regional clocks across various regions of the FPGA fabric.

Once the available clocking resources in the FPGAs are understood, we should determine what additional external clock sources and routing will properly exploit them and maintain the most flexibility for all future applications.

The following is a list of recommended considerations:

- Where is the clock sourced?
 - Generated locally on main board.
 - External source to the main board.
 - Generated in FPGA.
- What clock rates are required?
 - Estimate the range of possible FPGA clock frequencies.
 - Plan to generate an arbitrary clock rate, with fine granularity.
- What clock skew can be tolerated?
 - Inter-FPGA synchronization: make sure all FPGAs receive the clock sources at an acceptable skew.

- Inter-board synchronization: in large systems, make sure all clocks arrive at all FPGAs with an acceptable skew.

In addressing the above considerations, an in-house board might typically include most or all of the following elements:

- **On-board clock synthesis:** typically a PLL driven by a crystal oscillator reference with configurable parameters to select the desired clock frequency. To increase the flexibility the crystal oscillator may be removable. Multiple clocks generators may be needed to support systems with multiple clocks.
- **Input clock source selector:** given the multiple sources from which clocks can be sourced e.g., local (on board, FPGAs), or external, a clock source multiplexer should be implemented. Management of the multiplexer could be by manual switches or programmed by a separate software utility (see section 5.4 below).
- **Clock distribution:** regardless of the clock source, clock distribution must ensure the clock is distributed to local and external consumers with an acceptable skew. Excessive skew may result in improper logic propagation and decrease the timing margins between two or more entities passing synchronous signals to each other. On-board and connector delays must be accounted for and equalized while maintaining acceptable signal quality.
- **External clock source:** propagation delay from one board to another must be accounted for and propagation delays should be equalized by appropriate phase shifting. Special attention must be paid to a situation where multiple PLLs are cascaded between the source and destination of the clock path, as instability and loss of lock may occur if not properly designed.
- **Special high-speed clocks:** in addition to the application clocks, there may be a need for high-speed clocks to multiplex multiple signals on one pin. This is typically used when the number of signals between two FPGAs is greater than the number available pins between them. Using a high-speed clock the signals can be time-multiplexed over a single pin at the source and are then de-multiplexed at the receiving end. For this scheme to work properly both sides of the interface must have the same high-speed clock with minimal skew. Some prototyping tools such as Synopsys' CPM (Certify Pin Multiplier) assist the use of this technique.
- **Clock scaling:** Flexibility in clocking requires that the top-level clocks can be scaled to the correct frequencies for various parts of the design. Although dividers and other logic resources can be used for this inside the FPGA devices, PLLs also have their role to play.

5.4. Clock control and configuration

With the options discussed above, it is clear that the clock configuration of our board could become rather complicated. A large number of switches may be necessary to fully cover all the clocking options on a sophisticated multi-FPGA board. Nevertheless, with good documentation and the correct training, a board with such clocking flexibility would be more likely to be re-usable over multiple projects.

Alternatively, we might use programmable switches under the control of an on-board hosted microcontroller, or a utility running on an external host PC, as we shall explore in chapter 6.

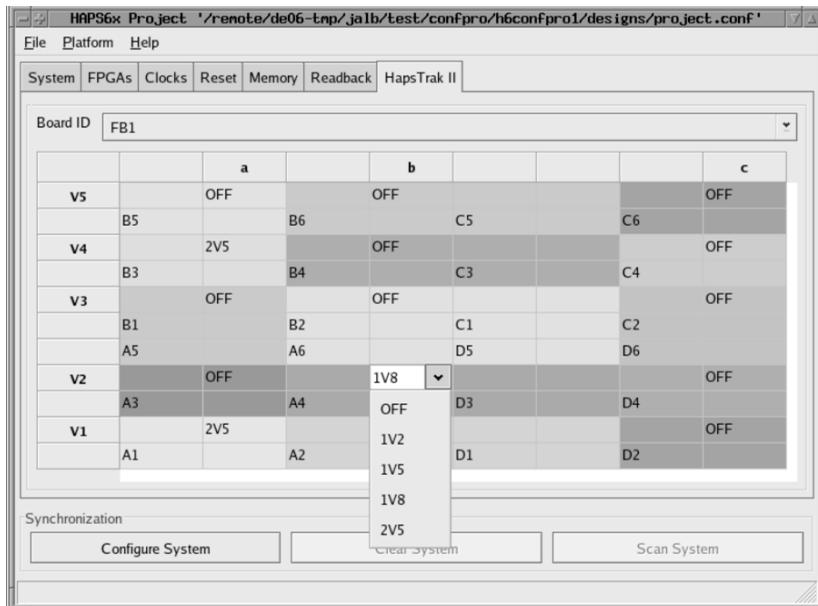
5.5. On-board Voltage Domains

SoC designs use an increasingly complex collection of voltages both for interfacing with external components and in an internal hierarchy of voltage islands in order to reduce power (see the ARM®-Synopsys Low-Power Methodology Manual, LPMM, in the bibliography for further detail). The internal voltage islands are usually inferred by EDA tools, based on the power-intent captured in side-files written in formats like the universal power format (UPF). Internal voltage islands therefore do not generally appear in the design RTL and do not get mapped onto the FPGA-based prototype. If the RTL delivered to the prototyping team already has voltage control logic, retention registers etc. then these will need to be tied to inactive state (see chapter 7). It is very unlikely that the voltage regions on the FPGA board will correspond with the regions within the SoC itself.

External voltage needs must still be mapped correctly, however. FPGA IO pins are arranged in banks that support different voltage thresholds and are powered by different voltage rings. It is important that the platform can supply any of the required voltages and route them to the relevant FPGA power pins. Voltage inflexibility additionally constrains IO usage and placement so that voltage rather than performance or connectivity sometimes governs partitioning choices with the result of compromising other priorities. Indeed, new voltage requirements can be a reason that existing boards cannot be reused in new projects, so building in flexibility is a good investment for helping to ensure longer life of the boards across multiple projects.

Figure 53 shows a screenshot from a utility which is used to remotely set-up the voltage of different regions on a HAPS-64 board.

Figure 53: On-board IO voltage configuration



Here we can see that the FPGA pins and daughter-card connectors are grouped into different regions which can be set to chosen voltages (in reality, there is good use of color to clarify the user interface). The region voltage is selected by use of pull-down menus and we can see that “OFF” is also a valid option.

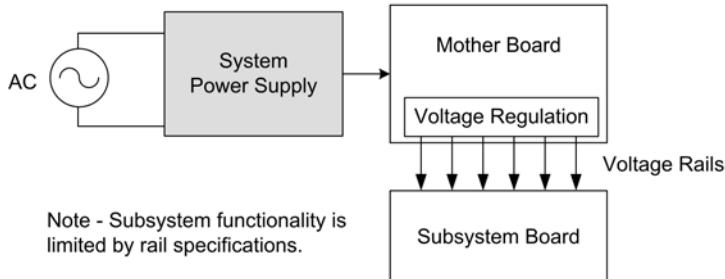
This configuration tool then configures programmable switches on the board which setup the reference and rail voltage for the FPGA devices themselves. Although this is an example from a commercial board, it is possible to create such utilities in-house and might be worth the effort if many copies of a board are to be made and used in a wide variety of projects.

5.6. Power supply and distribution

When considering the power supply scheme, we need to consider the system’s scalability, worst-case power budget, and planned modularity. For example, a multi-board system allows greater flexibility but the arrangement of the boards and sub-systems is impacted by their power requirements.

During system assembly to support daughter boards which draw power from the main board, as shown in the simple example of Figure 54, careful power budgeting for the additional boards must be done. If the combined current consumption on the daughter boards will exceed the available power on the main board, separate power must be supplied to the daughter board from the main external power supply as to not damage the main board supply circuit and possibly the board traces. In addition, to prevent damage to the main board from accidental power short on the daughter

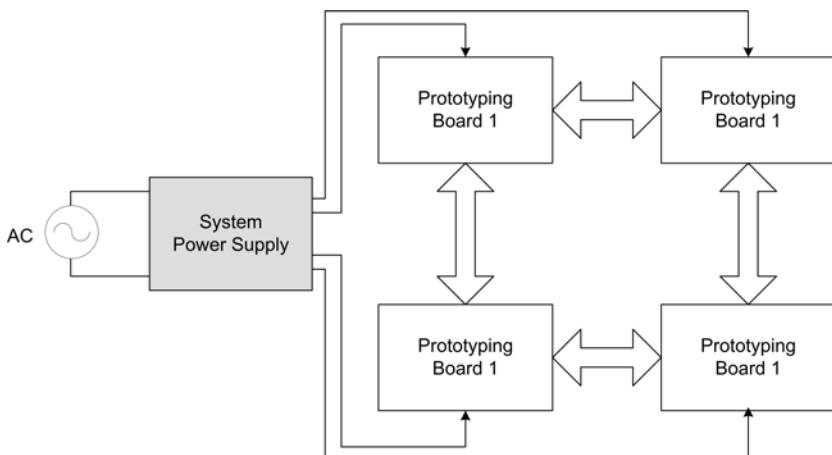
Figure 54: Example of non-recommended power distribution



board, a current limiting circuit, or a “resettable fuse” should be considered at every power exit point. The connector pins supplying the current should also be properly sized to the expected current draw for reliable and long-term system reliability.

It is bad practice for major sub-system boards to be powered from a host board, especially sub-system boards which mount their own FPGAs. It may be tempting to

Figure 55: Example of centralized power distribution



have voltage regulation and distribution from the host board onto all secondary boards, however, the latter may require current which exceeds the capability of board-to-board connectors carrying the voltage rails. This may be especially apparent during FPGA power-up and configuration, or even during live reset. Momentary brown-out of voltage rails, because of some particular activity of the FPGA, can be a hard problem to diagnose late in the prototyping project.

Far more preferable is that each board be specified with its own power input, and voltage regulation and distribution are managed locally. Each board then draws its input power from a common external power supply that is large enough to power all boards in the system as described in Figure 55..

As shown in the diagram, one power supply provides power to each of the boards in the system so it must be capable of sourcing power to all boards combined under worst-case scenario. Estimating the total power consumption is covered in the next section.

5.6.1. Board-level power distribution

Each FPGA board will need a number of voltages depending on the FPGA specifications and user selections:

- **Core voltage:** used to power the FPGA internal logic, is typically lower than the IO voltage and determined by the FPGA vendor.
- **IO voltage:** used to power the FPGA pins, and can vary depending on the signaling levels used in the system. IO voltages are typically grouped in banks that share the same IO voltage, so there is a potential to have multiple IO voltages across all IO pins. However, using multiple IO voltages will require multiple “power islands” in the PCB which will complicate the layout design. When selecting signaling levels, we need to consider the implication on IO speed, noise margin and possible interface constraints. All inter-FPGA connectivity can easily use the same signaling standard, but we need to make sure the signaling level on the FPGA side will be compatible with external devices to which the system may be connected.
- **IO current:** if the requirement of the externally connected device cannot be met using the FPGAs programmable output pin drives of up to 24mA, then we will need to add external buffers between the FPGA IOs and external components, probably mounted on daughter cards. It may only be necessary to add these buffers to a subset of the FPGA pins, if any.

On each board, the power may be supplied by single or multiple voltages, and then lower voltages are generated using power generation circuitry, such as DC/DC

converters. Given the multiple possible signaling standards, we may want to allow a certain degree of programmability of the IO voltages.

In estimating the power consumption, we should consider worst-case scenarios as the actual consumption depends on a number factors that may vary and are unknown for any arbitrary design in the future. Specifically, the power needed for each FPGA depends mostly on the switching rate and the number of internal FFs used in a design, so an estimation of these two factors must be made. In addition, IO switching and static current consumption should also be accounted for.

We must take into account initial power up requirements of the system because high current spikes often occur when powering up FPGAs (see chapter 11 for more details). Including all the above considerations, we can see that estimating total power is somewhat statistical but in all cases we should be conservative. To help estimate the power consumption for a given FPGA, vendors typically provide power estimation tools that make it easier to predict the power consumption for each FPGA.

Since core and some IO voltages are considerably lower than device voltages used in the past, the tolerance to variations is proportionately smaller as well. For proper FPGA operation, it is strongly recommended to employ some power monitoring circuits. Such devices will change state when voltage drops below a certain level, often programmable. When power faults occur, an alarm circuit should be triggered to alert the user to them. More about power faults management is described below.

In addition to the FPGA power needs, we should consider making provisions for additional programmable voltage level supplies to be used as reference voltages for auxiliary analog circuits such as ADC and DACs and other auxiliary circuits to be added to the FPGA system.

Recommendation: when connecting multiple boards together, it is safer to allocate a separate power distribution to each rather than have secondary boards which piggy-back onto a prime board. Piggy-back boards, sometimes called daughter boards or mezzanine boards, must meet a limited power spec in order not to overload the power supply of the host board.

5.6.2. Power distribution physical design considerations

The physical delivery of power to FPGAs is very critical to system stability and reliability. Since the FPGAs are the main power consumers, high-speed clocking and large numbers of FFs switching simultaneously will result in large current spikes at the clock edge where most logic changes take place. These current spikes, if not properly supplied, will cause power level drop or noise on the power lines at FPGA power pins and can result in unreliable operation.

To mitigate this potential problem, good low-impedance power planes should be used with adequate amount of continuous copper between the power supply circuit and the FPGA's power pins. Equally important is the power return path, or ground, so a low-impedance ground to the power supply is also necessary. In addition, large reservoir ($100\mu F$) and high-frequency small ($.1\mu F$) low-ESR, capacitors should be placed as close as possible to the power pins of the FPGAs in order to smooth the effects of both the slow and fast current surges.

In addition, special attention must be paid to IO power and grounding. Adequate power distribution should be implemented as mentioned above, but also care must be paid to signal groupings and ground pins. Typically, there is a ground pin per a number of IO pins. The number of IO pins per ground pin varies depending on the FPGA. Many pins switching at the same time to/from the same logic levels can result in a large return current through the ground pin. This may cause a “ground bounce,” where the ground reference rises momentarily and may cause incorrect logic interpretation in input pins associated with this ground pin. To mitigate this risk, it is advised to provide an adequate low impedance ground for the ground pins that are between the IO pins, and carefully follow the FPGA manufacturer’s board layout recommendations. In addition, to minimize the return current it is recommended to configure the FPGA IO such that they use the least amount of current drive on output pins rather than full 24mA on all pins by default. This will reduce current while maintaining adequate drive and signal quality.

5.7. System reliability management

As with any other design and test equipment, the reliability of the prototyping system is of critical importance to the user. A reliable system should have built-in protection from faults to the greatest extent possible and provide the user with warnings, should a fault occur. The types of faults typical to such large systems relate to power supply and thermal management.

5.7.1. Power supply monitoring

Proper operating voltage should be supplied to all FPGA and other components by design. However, power supply faults still may occur for variety of reasons, such as main power supply failure, unintended excessive current draw, or even accidental shorts on the power lines.

While such faults cannot be automatically repaired, it’s important the user is aware if such a fault has occurred. Therefore it is recommended that each power supply circuit has a power monitor circuit that continuously measures the voltage level and asserts a fault signal when the voltage is outside the monitoring range.

In addition to the on-board power monitoring, current generation FPGAs have a built-in system monitor circuit that can monitor all on-chip voltages and external voltages, as we shall see below. Since such voltage monitoring circuits are available “for free” in the FPGA it is recommended to use them.

Once a power failure is detected, the system management circuit can do one or all of the following:

- Set a visible indication of the fault, such as an LED. A more sophisticated system may have the fault monitoring circuit linked to a host computer, such that its status can be read by a program available to the user. Such a feature is critical when the system is housed in an enclosure, or located at a remote location away from the user.
- Assert a global reset signal to reduce operating current draw.
- De-configure all FPGAs, to ensure they consume the least amount of power. This will also serve the purpose of alerting a remote user that a system fault has occurred.

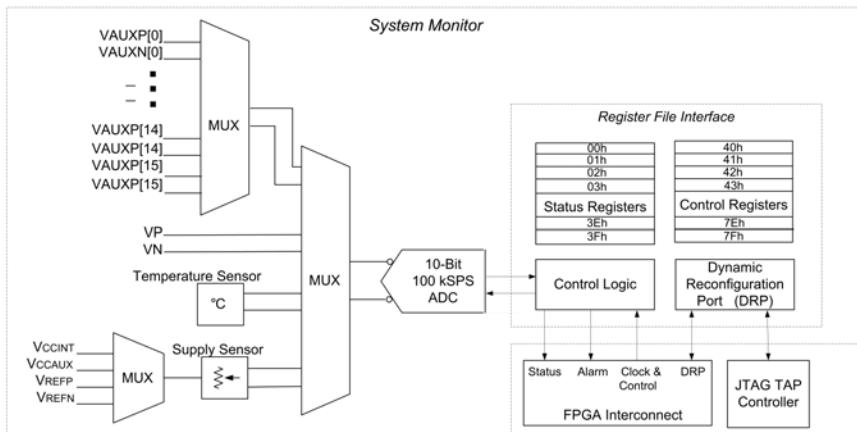
5.7.2. Temperature monitoring and management

As FPGA devices became larger and faster, so did their appetite for power, despite the shrinking node geometries. Current generation FPGAs with high utilization levels, running at high clock rates, may generate more heat than their package can dissipate to free air. Overheating die may result in malfunction and may cause irreversible damage to the FPGA and the PCB. In addition, when the prototyping system is enclosed in a box where the ventilation is marginal, FPGA temperature can rise to damaging levels. In either case thermal management circuitry should be considered and thermal requirements must take into account these worst case scenarios.

As introduced above, the FPGA’s system monitor” is highly programmable and has multiple modes of operation, but importantly it can also monitor the FPGA die’s average junction temperature. Monitored data is available at the JTAG port and also via the FPGA-resident design if the system monitor module is instantiated in the design. This could be a part of the FPGA-specific chip support block mentioned in chapter 4.

Figure 56 shows the system monitor circuit available at the core of the Virtex®-5 FPGA. As shown, in addition to the voltage monitoring, the system monitor has a temperature sensor with an associate alarm signal that is activated when the junction temperature exceeds its acceptable level. Such a signal can be used to implement a system-level power and temperature monitoring circuit, and alert the user to a fault in the system.

Figure 56: FPGA System Monitor



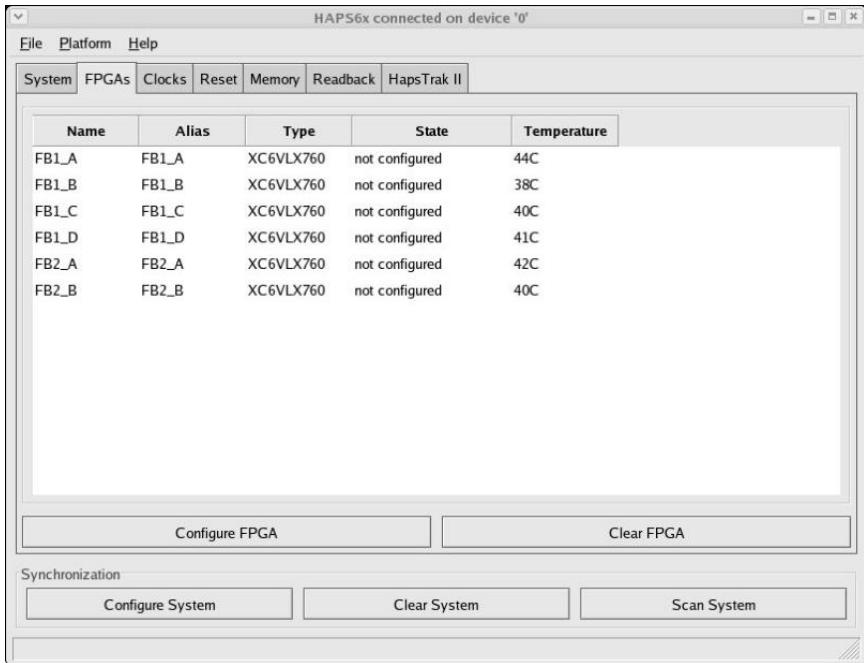
Copyright © 2011 Xilinx, Inc.

Once an over-temperature alarm is set, the system management circuit may do some or all of the following:

- Set a visible alarm, like an LED.
- Turn on fans to remove the excess heat.
- Put the system in reset condition.
- De-configure all FPGAs, to make sure they will consume the smallest amount of power.
- An example of how temperature could be monitored from a host is shown in Figure 57 . This is a screen shot of a PC-hosted utility called CONFPRO which is connected to a supervisor microcontroller running locally on a HAPS board (actually running on a Microblaze CPU embedded in an FPGA on the board). The screen shows the values read from the system monitors of six Virtex®-6 devices and any mitigation steps in progress e.g., controlling fans mounted on each FPGA. The same microcontroller also controls the FPGA clocks, resets and configuration so each of the above mitigation techniques can be automated locally or under control of the user.

In addition to the on-chip temperature monitoring, it's recommended the system also includes a number of temperature sensors placed close to where temperature is expected to be higher than other parts of the system, typically close to the FPGAs. Such devices are typically programmable and can produce an alarm signal when temperature is higher than their programmed value. Connecting these devices

Figure 57: monitor in CONFPRO



together and combining them with the on-chip temperature sensors/alarms will create a well monitored system.

5.7.3. FPGA cooling

To improve heat dissipation, it's recommended to install a heat sink to each FPGA. There are many such heat sinks that come with self-adhesive film, so installation is fairly simple. If a heat sink alone is not adequate enough to remove the heat from the FPGAs, a small fan over the heat sink or a fan with a heat sink combination can be placed on each FPGA to significantly improve the heat dissipation. To accommodate these fans, the necessary power supply and connectors should be incorporated into the main board. The fans need not be on all the time but could be controlled by a small loop making use of the FPGA's temperature monitor.

In the long run, boards run with FPGAs having not heat sink or fan cooling might receive temperature stress and prove to be less reliable.

5.8. FPGA configuration

FPGA configuration is the process in which the FPGAs are programmed with a bit stream and take on our intended functionality. Configuration data, which can range in the tens of Mbits per FPGA, is generated by the FPGA tools after place & route is completed.

Since FPGAs are SRAM-based volatile devices, the configuration process must take place after each power-up, but can also be performed an unlimited number of times after initial configuration.

There are a number of methods to configure FPGAs that can be broadly described as parallel or serial, master or slave, and JTAG. The mode is determined by hardwiring dedicated configuration mode signals. The following list describes the major characteristics of the different configuration modes:

- In the master modes, the FPGA is controlling the configuration upon power-up or when triggered by a configuration pin.
- In slave modes an external device controls the configuration interface.
- Serial configuration is slower than parallel but uses less signals, therefore leaving more signals to be used for the application itself.
- JTAG interface overrides and other configuration modes.

Configuration data can reside in a non-volatile memory or at a host computer. Multiple bit streams can be placed sequentially in the storage device and all FPGAs are configured in order in a sequence determined by hardwiring each FPGA.

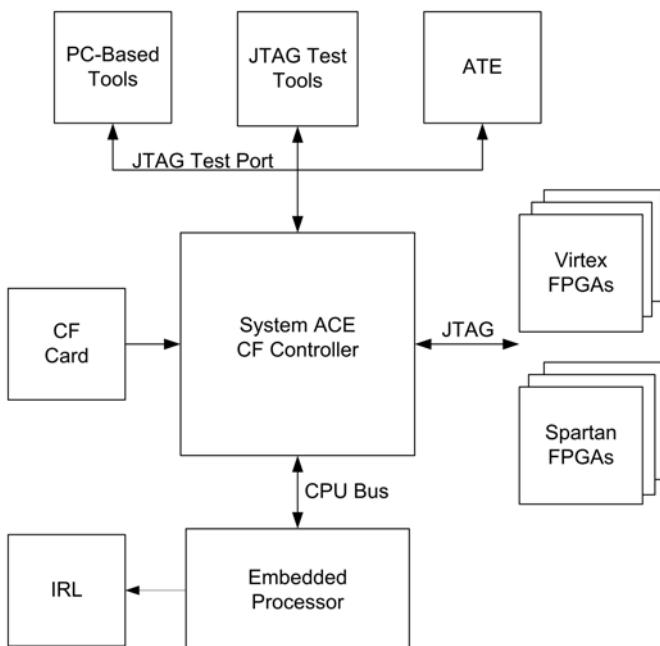
The most common configuration mode is via JTAG and this is usually how the download cables from host PC's communicates with the devices. However, it would be non-intuitive for those not familiar with FPGAs to make use of that download cable approach.

For non-FPGA experts, we can employ some added dedicated circuitry and common CompactFLASH™ (CF) cards. Commercial CF devices are readily available and of high enough capacity to hold the configuration data of even the largest FPGA devices. In fact, multiple configurations can be stored in the same card and the end-users can choose which one is loaded on the board by a simple setting of switches or even by remote control from a utility running on a host PC in the lab. There would still be room left over in the CF memory to hold other data, for example, documentation and user guides. This approach is very popular with those creating multiple copies of the FPGA platform for distribution to remote end-users. Removable CF cards provide the flexibility to transport the systems configuration to remote locations.

To enable the use of CF cards, a dedicated circuit called the Xilinx® System ACE™ controller is typically implemented in a separate small FPGA on the board. System

ACE technology is a piece of Xilinx® IP which reads the CF card and then emulates a JTAG interface memory for the FPGAs to be configured. Figure 58 shows a multi-mode configuration approach centered upon the System ACE™ technology

Figure 58: CompactFLASH™ based configuration via System ACE™ technology



Copyright © 2011 Xilinx, Inc.

More information about System ACE and for a full description of FPGA configuration modes, refer to Xilinx® documentation listed in the references.

5.9. Signal integrity

As mentioned above, clock skew should be minimized to an acceptable level. The main contributors to clock skew are jitter in the source signal, and noisy power supply and signal environment. To minimize these negative effects, special care must be paid to the signal propagation on the boards and across boards including

- Consistent impedance of all clock signals is critical.

- Proper routing on the board will reduce cross-talk with other signals.
- Well regulated power supplies with low-impedance paths to all loads.
- Trace length and delay matching for most likely parallel data and clock paths e.g., from an FPGA to its dedicated IO connector.
- There are other sources for understanding the issues on board layout and signal integrity (see references). This is not a manual for PCB layout and design and we only mention it here as a reminder that creating a large multi-FPGA board is a challenging problem, although not insurmountable with access to the right PCB-layout expertise.

5.10. Global start-up and reset

Once a system clock generation and distribution scheme is designed, we need to consider the effects of integrating the clocking circuits with the FPGA system. In a multi-FPGA prototype in particular, it is essential that all FPGAs become operational at the same time i.e., have their resets de-asserted on the same clock edge.

When PLLs are used either on the board and/or inside the FPGAs, the system start-up must be delayed from after power-up until all timing circuits (PLLs) are stable and phase locked to their respective sources.

In this case, we employ a global reset circuit that holds the whole system at the reset state until all PLLs in the system are locked. There is more detail on this in chapter 8, however, during board development, it is important to make external PLL device locked an reset signals available at the pins of at least one FPGA so that they might be gated into an overall reset condition tree.

5.11. Robustness

As mentioned in chapter 2, one excellent reason to use FPGA-based prototypes during SoC verification is to allow the design to be taken out of the lab. To enable such use, platforms must be first of all reliable but that is also true of in-lab platforms. For portability, the platform must also be robust enough to survive non-lab conditions. It is important that the platform is able to work as well and as error-free as any other piece of field test equipment.

The platform may be made more robust by using strengthening struts to add rigidity to the board itself, by securing daughter cards using spacers, pillars and screws. The whole assembly may also be held in a strong case or rack but if so, then consideration should be given to maintaining the necessary access to test points,

extension card sites and peripheral ports. The case should also provide for the full power and cooling needs of the platform in the field. Early consideration of these points may avoid frequent remedial action when the platforms are in the field or used by others outside the lab, such as software developers, for whom the platform is just a box under their desk and should be as simple and reliable as possible.

We will revisit the subject of robustness in more detail in chapter 12 where we consider how we might take our prototype out of the lab and into the field for various reasons.

5.12. Adopting a standard in-house platform

So far, we have given long consideration of the topology of the platform and its potential for reuse in follow-on projects. R&D centers where FPGA-based prototyping is widely used tend to adopt a standardized connectivity and modularity. Various experienced prototyping teams have adopted a tile-based approach where each new board, at whichever R&D site it was developed, is designed and built to a common form-factor and using common connectors and pin-out. This allows the potential for each board to be reused by other sites, increasing the potential return on that investment. Such a standardized approach does however take considerable co-ordination and management and can lead to constraints on new platforms in order to fit in with the legacy of the standard. As new boards are developed, the choice will often arise between staying within the standard and making a specific departure in order to better meet the needs of the current project. For example, should a given IO socket be added to the platform, even though it would prevent or limit physical connections to other standard modules?

Legacy can also be a significant obstacle during step-changes of in-house design practice, for example, when switching to a new wider SoC bus which may overflow the standard connector interface between boards.

An important part of an in-house standard is the physical connectivity. Tile-based or stack-based arrangements have allowed boards built on the standard to be connected together in a multitude of three-dimensional arrangements. Some of these standard tiles do not have FPGA resources at all but instead hold peripherals or bespoke external interfaces and connectors, for example SCART sockets or IO standard interfaces. This meets the modularity guidelines mentioned earlier in this chapter and aids reuse but distributing components over a large area can have an effect upon overall performance. The authors have seen a number of examples of stacking modular systems where performance and reliability have proved inadequate so the connectors between modules become a critical part of any in-house standard platform. The connectors must not only carry enough signals, they must also have good electrical characteristics, be high-quality and physically robust. Choosing connectors on price will prove to be a false economy in the long run.

Nevertheless, the advantages of an in-house standard are significant when matched with a suitable infrastructure for continued development, distribution, documentation and support. Let's quickly consider each of these in turn:

- **Development:** to prevent duplicate development effort, for example multiple but similar boards containing the same FPGA, it is necessary to coordinate FPGA-based prototyping specialists over a number of sites and projects towards supporting and growing the in-house standard. An infrastructure is required which includes a catalog of available boards, their specifications, technical documentation and the roadmap/schedule for new board development.
- **Distribution:** those entering into new FPGA-based prototyping projects need to be able to specify, source and assemble the required platform quickly from available boards. Only if boards are not immediately or quickly available should a new board development be considered. To make that possible, board developers and manufacturing sources need to have visibility of forthcoming projects so that the necessary boards can be created and put into stock. The management of the in-house standard becomes a sophisticated business within a business. It will take considerable resources to maintain but will reduce reliance on single contributors and help to protect the investment on previous in-house platform development.
- **Documentation:** the standard itself must be documented and an arbiter appointed to certify the compliance of any new board. Then each individual board should have sufficient documentation to firstly allow other users to decide its suitability for a project but then to fully describe every aspect of its final use. It may also be beneficial to create training material and reference design examples to shorten the learning curve for new users, some of whom will be remote from the original board designers. Insufficient documentation will not only result in delayed projects and inefficient use of the board but also require more help from support personnel.
- **Support:** the in-house standard will need personnel to support the use of boards within FPGA-based prototyping platforms, often in many sites and projects at the same time. Modern intranet and other internal media make this more possible than even a few years ago but it may still require that large installations of platforms would require accompanying on-site support personnel, depending on platform reliability and ease-of-use. The support infrastructure might be centralized or distributed but must certainly have continuous availability and expertise, minimizing the risk to individual projects of key support resources becoming unavailable. A natural extension of support for the platform is the creation of a central lab for FPGA-based prototyping, offering not only the platform but also the

design services, expertise, scheduling and management for the entire project. An SoC team may then choose to outsource the entire FPGA-based prototype as a package of work within the overall SoC project budget and schedule. This has become a popular option for many FPGA-based prototyping projects and falls under the normal rules for economy of scale. It is interesting that there are such centralized design service teams who do not actually make their own boards but instead assemble platforms from ready-made boards from external suppliers.

To summarize, the establishment of an in-house standard *platform* can require a large investment in time, expertise and personnel. Historically, the scale of this infrastructure has meant that in-house standards were generally only adopted by larger multi-site corporations, performing regular prototyping projects. However, as we shall see in chapter 6, the growth of commercial platform suppliers has meant this infrastructure can be largely outsourced and now reuse and standardization is available to all teams, whether at a corporation or a start-up.

Having spent chapter 5 exploring the detailed considerations in creating an FPGA-based hardware platform for an FPGA-based prototyping project, we will now consider the other side of the so called “make versus buy” argument. When does it make more sense to use a ready-made FPGA board or even a more sophisticated FPGA-based system instead of designing a bespoke board? What interconnect or modularity scheme is best for our needs? What are our needs anyway, and will they remain the same over many projects? The authors hope that this short chapter will compliment chapter 5 and allow readers to make an informed and confident choice between the various platform options.

6.1. What do you need the board to do?

The above subtitle is not meant to be a flippant, rhetorical question. When investigating ready-made boards or systems, it is possible to lose sight of what we really need the board to do; in our case, the boards are to be used for FPGA-based prototyping.

There are so many FPGA boards and other FPGA-based target systems in the market at many different price and performance points. There are also many reasons why these various platforms have been developed and different uses for which they may be intentionally optimized by their developers. For example, there are FPGA boards which are intended for resale in low-volume production items, such as the boards conforming to the FPGA mezzanine card (FMC) standard (part of VITA 57). These FMC boards might not be useful for SoC prototyping for many reasons, especially their relatively low capacity. On the other hand, there are boards which are designed to be very low-cost target boards for those evaluating the use of a particular FPGA device or IP; once again, these are not intended to be FPGA-based prototyping platforms as we describe them in this book but may nevertheless be sold as a “prototyping board.”

Before discriminating between these various offerings, it is important to understand your goals and to weigh the strengths and weaknesses of the various options accordingly.

As discussed in chapter 2, the various goals of using an FPGA board might include:

- Verification of the functionality of the original RTL code
- Testing of in-system performance of algorithms under development
- Verification of in-system operation of external IP
- Regression testing for late-project engineering change orders (ECO)
- In-system test of the physical-layer software and drivers
- Early integration of applications into the embedded operating system
- Creation of stand-alone target platforms for software verification
- Provision of evaluation or software development platforms to potential partners
- Implementing a company-wide standard verification infrastructure for multiple projects
- Trade-show demonstrations of not-yet realized products
- Participation of standards organization's "plugfest" compatibility events

Each of these goals place different constraints on the board and it is unreasonable to expect any particular board to fulfill optimally all goals listed above. For example, a complex board assembly with many test points to assist in debugging RTL may not be convenient or robust enough for use as a stand-alone software target. On the other hand, an in-house board created to fit the form factor and price required for shipment to many potential partners may not have the flexibility for use in derivative projects.

6.2. Choosing the board(s) to meet your goals

The relative merits of the various prototyping boards are subjective and it is not the purpose of this chapter to tell the reader which board is best. Acknowledging this subjectivity, Table 11 offers one specific suggestion for the weighting that might be applied to various features based on the above goals. For example, when creating a large number of evaluation boards, each board should be cheap and robust. On the other hand, when building a standard company infrastructure for multiple SoC prototyping projects, flexibility is probably more important.

We do not expect everybody to agree with the entries in the table, but the important thing is to assess each offering similarly when considering our next board investment. There are a large number of boards available, as mentioned, and we can become lost in details and literature when making our comparisons. Having a checklist such as that in table 11 may help us make a quicker and more appropriate

choice. The column for cost may seem unnecessary (i.e., when is cost NOT important?) but we are trying to warn against false economy while still recognizing that in some cases, cost is the primary concern.

Readers may find it helpful to recreate this table with their own priorities so that all stakeholders can agree and remain focused upon key criteria driving the choice

Table 11: Example of how importance of board features depends on intended use

Intended use	Degree of Criticality of Various Board Factors						
	Flexibility	Speed	Debug	Robustness	Capacity	Cost	Delivery
Key: +++ very critical, - - - irrelevant, = depends on other factors							
1 RTL debug	++	++	+++	=	=	=	++
2 Algorithm research	+	=	++	=	-	=	=
3 IP evaluation	=	++	+	=	=	=	=
4 Regression test for ECO	+	++	=	+	+	=	++
5 Driver test	=	++	+	+	-	=	=
6 App/OS integration	=	+	-	++	++	=	=
7 Software verification	-	+	-	++	+	+	=
8 Eval boards	--	=	+	+++	--	++	++
9 Standard infrastructure	+++	+	++	+	++	=	+

For the rest of this chapter we will explore the more critical of the decision criteria listed as column headings in Table 11.

6.3. Flexibility: modularity

One of the main contributors towards system flexibility is the physical arrangement of the boards themselves. For example, there are commercial boards on the market which have 20 or more FPGAs mounted on a single board. If you need close to 20 FPGAs for your design, then such a monster board might look attractive. On the other hand, if you need considerably fewer FPGAs for this project but possibly more for the next one, then such a board may not be very efficient. A modular system that allows for expansion or the distribution of the number of FPGAs can yield greater return on investment because it is more likely that the boards will be reused over multiple projects. For example, an eight-FPGA prototype might fit well onto a ten-FPGA board (allowing room for those mid-project enhancements) but using a modular system of two four-FPGA boards would also work assuming extra boards can be added. The latter approach would allow a smaller follow-on project to use each of the four-FPGA boards autonomously and separately whereas attempting to reuse the former ten-FPGA board would be far less efficient.

This exact same approach is pertinent for the IO and peripherals present in the prototype. Just because a design has four USB channels, the board choice should not be limited only to boards that supply that number of channels. A flexible platform will be able to supply any number up to four and beyond. That also includes zero i.e., the base board should probably not include any USB or other specific interfaces but should readily allow these to be added. The reason for this is that loading the base-board with a cornucopia of peripheral functions not only wastes money and board area, most importantly it ties up FPGA pins which are dedicated to those peripherals, whether or not they are used.

In particular, modular add-ons are often the only way to support IP cores because either the RTL is not available or because a sensitive PHY component is required. These could obviously not be provided on a baseboard so either the IP vendor or the board vendor must support this in another way. In the case of Synopsys®, where boards and IP are supplied by the same vendor, some advantage can be passed on to the end user because the IP is pre-tested and available for a modular board system. In addition, as IP evolves to meet next-generation standards, designers may substitute a new add-on IP daughter card for the new standards without having to throw away the rest of the board.

An important differentiator between FPGA boards, therefore, is the breadth of available add-on peripheral functions and ease of their supply and use. An example of a library of peripheral boards can be seen at the online store of daughter cards supplied by Synopsys to support FPGA baseboards. The online store can be seen at www.synopsys.com/apps/haps/index .

In general, end-users should avoid a one-size-fits-all approach to selecting an FPGA-based prototyping vendor because in the large majority of cases, this will involve design, project and commercial compromise. Customers should expect their

vendors to offer a number of sizes and types of board; for example, different numbers of FPGAs, different IO interfaces etc. but it is important that they should be as cross-compatible as possible so that choosing a certain board does not preclude the addition of other resources later. The approach also requires that modules are readily available from the vendor's inventory, as the advantage of modularity can be lost if it takes too long to obtain the modules with which to build your platform.

A critical issue with such a modular approach is the potential for loss of performance as signals cross between the various components and indeed, whether or not the components may even be linked together with enough signals. We should therefore now look closely at the interconnect issues involved in maintaining flexibility and performance.

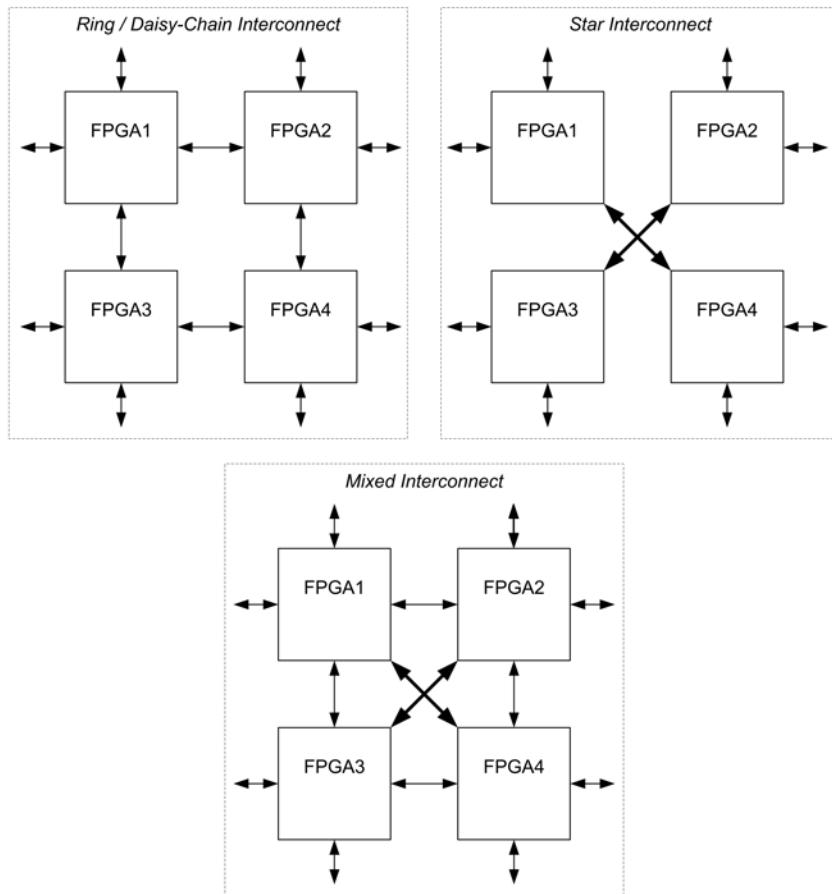
6.4. Flexibility: interconnect

As experienced prototypers will know, the effective performance and capacity of a FPGA-based prototype is often limited by the inter-FPGA connections. There is a great deal of resource inside today's FPGAs but the number of IO pins is limited by the package technology to typically around 1000 user IO pins. The 1000 pins need then be linked to other FPGAs or to peripherals on the board to make an interconnect network which is as universally applicable as possible, but what should that look like? Should the pins be daisy-chained between FPGAs in a ring arrangement or should all FPGA pins be joined together in a star? Should the FPGAs be linked only to adjacent FPGAs, or should some provision be made to link to more distant devices? These options are illustrated in Figure 59.

There are advantages and disadvantages to each option. For example, daisy-chained interconnect requires that signals be passed through FPGAs themselves in order to connect distant devices. Such pass-through connections not only limit the pin availability for other signals, but they also dramatically increase the overall path delay. Boards or systems that rely on pass-through interconnections typically run more slowly than other types of board. However, such a board might lend itself to a design which is dominated by a single wide datapath with little or no branching.

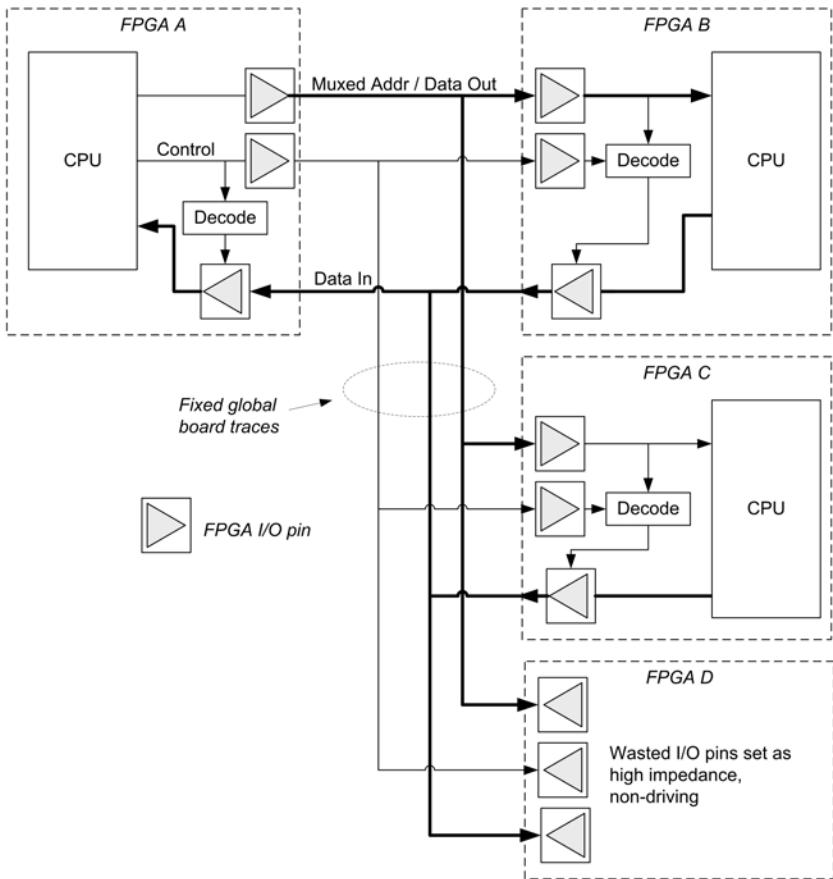
On the other hand, use of a star-connection may be faster because any FPGA can drive any other with a direct wire, but star-connection typically makes the least efficient use of FPGA pins. This can be seen by reconsidering the example we saw earlier in chapter 5, shown here again in Figure 60. Here we see how three design blocks on a simple multiplexed bus are partitioned into three out of the four FPGAs on a board, where the board employs star-based fixed interconnect.

Figure 59: Examples of fixed interconnect configurations found on FPGA boards



The connections between the three blocks are going to be of high-speed, owing to the direct connections, but many pins on the fourth FPGA will be wasted. Furthermore, these unused pins will need to be configured as high-impedance in order to not interfere with the desired signals. If the fourth FPGA is to be used for another part of the design, then this pin wastage may be critical and there may be a large impact on the ability to map the rest of the design into the remaining resources.

Figure 60: Multiplexed bus partitioned into a star-based interconnect.



In the case that a board is designed and manufactured in house, we can arrange interconnect exactly as required to meet the needs of the prototype project. The interconnection often resembles the top-level block diagram of the SoC design, especially when a Design-for-Prototype approach has been used to ease the partitioning of top-level SoC blocks into FPGAs. This freedom will probably produce an optimal interconnect arrangement for this prototyping project but is likely to be rather sub-optimal for follow-on projects. In addition, fixing interconnect resources at the start of a prototyping project may lead to problems if, or when, the SoC design changes as it progresses.

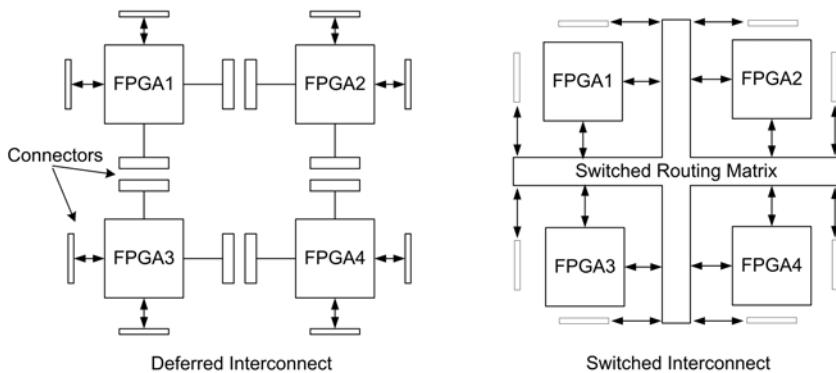
In the same way, a commercial board with fixed interconnect is unlikely to match the exact needs of a given project and compromise will be required. A typical

solution found on many commercial boards is a mix between the previously mentioned direct interconnect arrangements, but what is the best mix and therefore the best board for our design?

6.5. What is the ideal interconnect topology?

This brings us back to our decision criteria for selecting boards. Perhaps we are seeking as flexible interconnect arrangement as possible, but with high-performance and one that can also be tailored as closely as possible to meet the interconnect needs of a given prototyping project. Board vendors should understand the compromise inferred by these apparently contradictory needs and they should try to choose the optimal interconnect arrangement which will be applicable to as many end-users' projects as possible.

Figure 61: Two examples of Indirect Interconnect



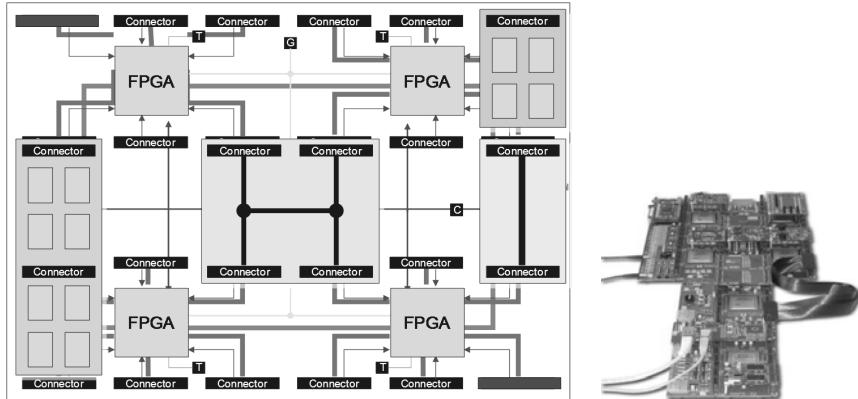
The most flexible solution is to use some form of indirect interconnect, the two most common examples of which are shown in Figure 61. The two examples are deferred interconnect and switched interconnect.

In a deferred interconnect topology, there are relatively few fixed connections between the FPGAs but instead each FPGA pin is routed to a nearby connector. Then other media, either connector-boards or some kind of flexible cables are used to link between these connectors as required for each specific project. For example, in Figure 61 we see a connector arrangement in which a large number of connections could be made between FPGA1 and FPGA4 by using linking cables between multiple connectors at each FPGA. Stacking connectors would still allow star-connection between multiple points if required.

One example of such a deferred interconnect scheme is called HapsTrak® II and is universally employed on the HAPS® products, created by the Platform Design Group of Synopsys at its design centre in Southern Sweden.

A diagram and photograph of a HAPS-based platform is seen in Figure 62, showing local and distant connections made with mezzanine connector boards and ribbon cables.

Figure 62: Synopsys HAPS®: example of deferred interconnect



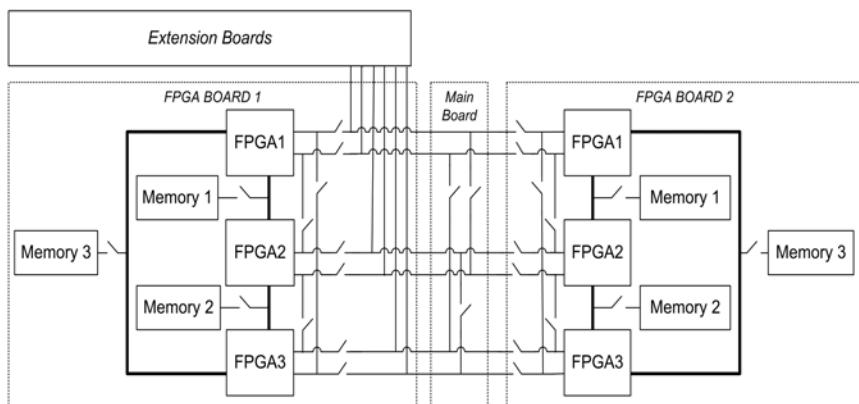
The granularity of the connectors will have an impact on their flexibility, but deferred interconnect offers a large number of possibilities for connecting the FPGAs together and also for connecting FPGAs to the rest of the system. Furthermore, because connections are not fixed, it is easy to disconnect the cables and connector boards and reconnect the FPGAs in a new topology for the next prototyping project.

The second example of indirect interconnect is switched interconnect, which relies on programmable switches to connect different portions of interconnect together. Of course, it might be possible to employ manual switches but the connection density is far lower and there is an increased chance of error. In the Figure 61 example, we have an idealized central-switched routing matrix which might connect any point to any point on the board. This matrix would be programmable and configured to meet the partitioning and routing needs for any given design. Being programmable, it would also be changed quickly between projects or even during projects to allow easy exploration of different design options. For remote operation, perhaps into a software validation lab at another site, a design image can be loaded into the system and the switched interconnect configured without anybody needing to touch the prototype itself.

In reality, such a universal cross-connect matrix as shown in Figure 61 is not likely to be used because it is difficult to scale to a large number of connections. Vendors will therefore investigate the most flexible trade-off between size, speed and flexibility in order to offer attractive solutions to potential users. This will probably involve some cascading of smaller switch matrices, mixed with some direct interconnections.

Another significant advantage of a switched interconnect approach is that it is very quick and easy to change, so that users need not think of the interconnect as static but something rather more dynamic. After the design is configured onto the board(s), it is still possible to route unused pins or other connections to other points, allowing, for example, quick exploration of workarounds or debug scenarios, or routing of extra signals to test and debug ports. In sophisticated examples, it is also possible to add debug instrumentations and links to other verifications technologies, such as RTL simulation. These latter ideas are discussed later in this section.

Figure 63: Switched interconnect matrix in CHIPit prototyping system



One further advantage of switched interconnect is that the programming of the switches can be placed under the control of the design partitioning tool. In this case, if the partitioner realizes that, to achieve an optimum partition, it needs more connections between certain FPGAs, it can immediately exercise that option while continuing the partitioning task with these amended interconnect resources. In these scenarios, a fine-grain switch fabric is most useful so that as few as necessary connections are made to solve the partitioning problem, while still allowing the rest of the local connections to be employed elsewhere.

Examples of such a switched interconnect solution are provided by the CHIPit® systems created in the Synopsys Design Centre in Erfurt, Germany. A CHIPit system allows the partitioner to allocate interconnections in granularity of eight paths at a time. An overview diagram of a CHIPit system switch-fabric topology is

shown in Figure 63. Here we can see that in addition to fixed traces between the FPGAs on the same board, there are also programmable switches which can be used to connect FPGAs by additional traces. Similar switches can link memory modules into the platform or link to other boards or platforms, building larger systems with more FPGAs. Not all combinations of switches are legal and they are too numerous to control manually in most cases, therefore they are configured automatically by software, which is also aware of system topology and partition requirements

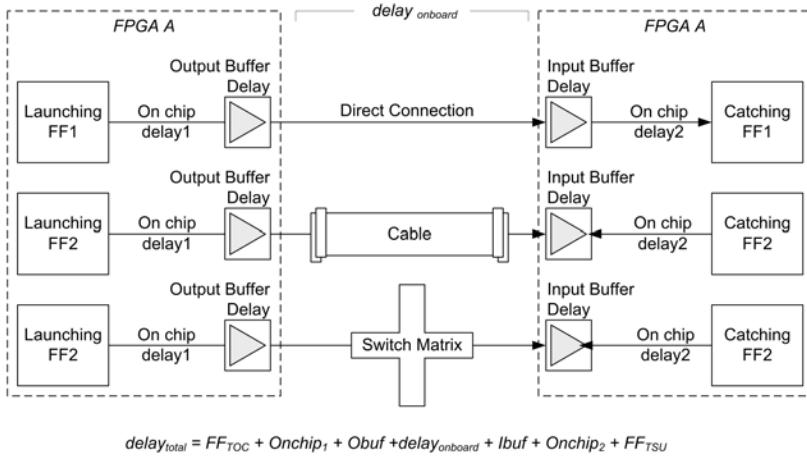
6.6. Speed: the effect of interconnect delay

It should be mentioned that the disadvantage of either of the indirect interconnect methods mentioned above is that the path between FPGA pins is less than ideal for propagating high-speed signals. In general, the highest performance path between two pins will be that with the lowest number of discontinuities in the path. Every additional transition such as a connector, solder joint and even in-board via adds some impedance which degrades the quality of the signal and increases its overall opportunity for crosstalk between signals. Upon close analysis, however, we find that this is not such a large problem as may be thought.

For the fastest connections, flight time is low and the majority of interconnect delay is in the FPGA pins themselves. It takes longer for signals to get off-chip and then on-chip than it takes them to travel across the relevant board trace. As prototype boards and the designs running on them become faster, however, this path delay between FPGAs will need to be mitigated in some way to prevent it from becoming critical. For that reason, Synopsys has introduced an automated method for using the high-speed differential signaling between FPGAs, employing source-synchronous paths in order to allow faster speed between FPGAs. We will discuss this technology a little further later in chapter 8.

Each of the indirect interconnect approaches adds some delay to the path between the FPGA pins compared to direct interconnect, but how significant is that? If we analyze the components of the total delay between two points in separate FPGAs, using three types of interconnect, then we discover paths resembling those in Figure 64.

Figure 64: Comparing path delays for different interconnect examples



Readers may argue that the FFs should be implemented in the IO pads of the FPGA. This is true, however, it is often not possible to do this for all on-chip/off-chip paths and so the worst-case path between two FPGAs may be non-ideal and actually starting and/or ending at internal FFs.

In this analysis, the path delay is given by the expression shown also in Figure 64. Common to each of the interconnect methods is a component of delay which is the total of all in-chip delays and on-chip/off-chip delays, expressed as:

$$\sum FF_{TOC}, FF_{TSU}, Onchip_1, Onchip_2, Obuf, Ibuf$$

where . . .

FF_{TOC} is the clock-to-output delay of the launching FF

FF_{TSU} is the set-up time of the catching FF

$Onchip_{1,2}$ are the routing delays inside the relevant FPGAs

$Obuf$ is the output pad delay of the first FPGA

$Ibuf$ is the input pad delay on the second FPGA

Using figures from a modern FPGA technology, such as the Xilinx® Virtex®-6 family, these delays can be found to total around 4.5ns. This time will be common to all the interconnect arrangements so we now have a way to qualify the impact of the different interconnects on overall path delay i.e., how do the different on-board components of the paths compare to this common delay of 4.5ns?

Firstly, direct interconnection on a well-designed high-quality board material results in typical estimates for $delay_{onboard}$, of approximately 1ns. We can consider this as

the signal “flight time” between FPGA pins. So already we see that even in direct interconnect, the total delay on the path is dominated by factors other than its flight time. Nevertheless, maximum performance could be as high as 250MHz if partitioning can be achieved which places the critical path over direct interconnect.

Deferred interconnect via good-quality cables or other connecting media, using highest quality sockets, results in typical flight time of approximately 4ns, which is now approximately the same as the rest of the path. We might therefore presume that if we use deferred interconnect and the critical path of a design traverses a cable, then the maximum performance of the prototype will be approximately half that which might be achieved with direct interconnect, i.e., approximately 125MHz instead of 250MHz.

However, raw flight time is seldom the real limiting factor in overall performance.

Supporting very many real-life FPGA-based prototyping projects over the years at Synopsys, we have discovered that there are performance-limiting factors beyond simple signal flight time. Commonly the speed of a prototype is dependent on a number of other factors as summarized below:

- The style of the RTL in the SoC design itself and how efficiently that can be mapped into FPGA
- Complexity of interconnection in the design, especially the buses
- Use of IP blocks which have no FPGA equivalent
- Percentage utilization of each FPGA
- Multiplexing ratio of signals between FPGAs
- Speed at which fast IO data can be channeled into the FPGA core

Completing our analysis by considering switched interconnect, we see that the delay for a single switch element is very small, somewhere between 0.15ns and 0.25ns, but the onboard delays to get to and from the switch must also be included, giving a single-pass flight time of approximately 1.5ns, best-case. In typical designs, however, flight time for switched interconnect is less easy to predict because in order to route the whole design, it will be necessary for some signals to traverse multiple segments of the matrix, via a number of switch elements. On average there are two switch traversals but this might be as high as eight in extreme case of a very large design which is partitioned across up to 20 FPGA devices. To ensure that the critical path traverses as few switches as possible, the board vendor must develop and support routing optimization tools. Furthermore, if this routing task can be under the control of the partitioner then the options for the tools become almost infinitely wider. Such a concurrent partition-and-route tool would provide the best results on a switched interconnect-based system so once again we see the benefit of a board to be supplied with sophisticated support tools.

Returning to our timing analysis, we find that a good design partition onto a switched interconnect board will provide typical flight times of 4ns, or approximately the same as a cable-based connection on a deferred interconnect

Table 12: Comparing raw path speed for different interconnect schemes.

Interconnect	Flight Time	Total Path	approx. Fmax
Direct	1ns	5.5ns	180MHz
Deferred	4ns	9.5ns	105MHz
Switched	4ns	9.5ns	105MHz

board, once again providing a top-speed performance over 100MHz. The results of the above analysis are summarized in Table 12.

6.6.1. How important is interconnect flight time?

In each of the above approximations, the performance of a prototype is assumed to be limited by the point-to-point connection between the FPGAs, but how often is that the case in a real project?

Certainly, the performance is usually limited by the interconnect, but the limit is actually in its availability rather than its flight time. In many FPGA-based prototyping projects, partitioning requires more pins than are physically provided by any FPGA and so it becomes necessary to use multiplexing in order to provide enough paths. For example, if the partition requires 200 connections between two particular FPGAs when only 100 free connections exist, then it doesn't matter if those 100 are direct or via the planet Mars; some compromise in function or performance will be necessary. This compromise is most commonly achieved by using multiplexing to place two signals on each of the free connections (as is discussed below). A 2:1 multiplexing ratio, even if possible at maximum speed on the 100 free connections, will in effect half the real system performance as the internal logic of the FPGAs will necessarily be running at half the speed of the interconnect, at most.

An indirect interconnect board will provide extra options and allow a different partition. This may liberate 100 or more pins which can then be used to run these 200 signals at full speed without multiplexing. This simple example shows how the flexibility of interconnect can provide greater real performance than direct interconnect in some situations and this is by no means an extreme example. Support for multiplexing is another feature to look for in an FPGA board; for example, clock multipliers for sampling the higher-rate multiplexed signals.

In the final case above, real-time IO is an important benefit of FPGA-based prototyping but the rest of the FPGA has to be able to keep up! This can be achieved by discarding some incoming data, or by splitting a high-speed input data stream into a number of parallel internal channels. For example, splitting a 400MHz input channel into eight parallel streams sampled at 50MHz. Fast IO capability is less valuable if the board or its FPGAs are sub-performance.

These limiting factors are more impacted by interconnect flexibility than by flight time, therefore, in the trade-off between speed and flexibility, the optimum decision is often weighted towards flexibility.

Summarizing our discussion on interconnect, the on-chip, off-chip delays associated with any FPGA technology cannot be avoided, but a good FPGA platform will provide a number of options so that critical signals may be routed via direct interconnect while less critical signals may take indirect interconnect. In fact, once non-critical signals are recognized, the user might even choose to multiplex them onto shared paths, increasing the number of pins available for the more critical signals. Good tools and automation will help this process.

6.7. Speed: quality of design and layout

As anyone that has prototyped in the past will attest, making a design run at very high speed is a significant task in itself, so we must be able to rely on our boards working to full specification every time. If boards delivered by a vendor show noticeable difference in performance or delay between batches, or even between boards in the same batch, then this is a sign that the board design is not of high quality.

For example, for an interface to run at speeds over 100MHz, required for native operation of interfaces such as PCIe or DDR3, the interface must have fast pins on its FPGAs and robust design and layout of the PCB itself. To do this right, particularly with the latest, high pin-count FPGAs, requires complex board designs with very many layers. Very few board vendors can design and build 40-layer boards for example. The board traces themselves must be length and impedance matched to allow good differential signaling and good delay matching between remote synchronous points. This will allow more freedom when partitioning any given design across multiple FPGAs.

This need for high-quality reproducible board performance is especially true of clock and reset networks, which must not only be flexible enough to allow a wide variety of clock sources and rates, they must also deliver good clock signal at every point in a distributed clock network.

Power supply is also a critical part of the design, and low impedance, high-current paths to the primary FPGA core and IO voltage rail pins are fundamental to

maintaining low noise, especially on designs which switch many signals between FPGAs on every clock cycle.

On first inspection, two boards that use the same FPGAs might seem to offer approximately the same speed and quality but it is the expertise of the board vendor that harnesses the raw FPGA performance and delivers it reliably. For example, even device temperature must be monitored and controlled in order to maintain reliability and achieve highest performance possible within limits of the board fabric. Let us look at another aspect of performance, that of trading off interconnect width versus speed using multiplexing.

6.8. On-board support for signal multiplexing

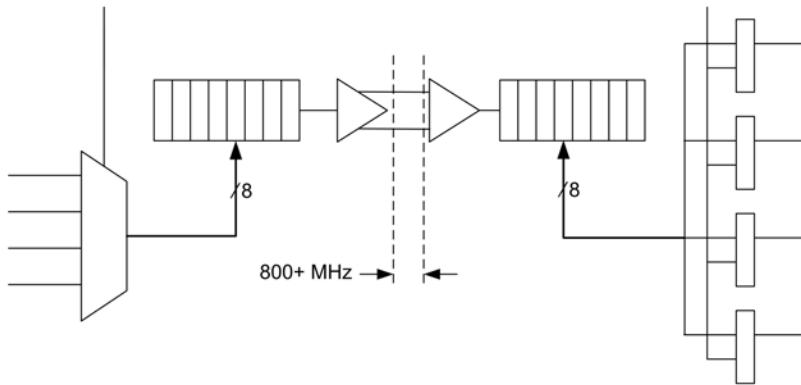
The concept of time division multiplexing (TDM) and its ability to increase effective IO between FPGAs is well understood and it is not difficult to see how two or more signals may share the same interconnect path between FPGA pins. The TDM approach would need mux and de-mux logic inside the FPGA and would need a way to keep the two ends synchronized. It is also necessary to run the TDM paths at a higher rate than the FPGA's internal logic, and also ensure that signals arriving at the mux or leaving the de-mux both meet the necessary timing constraints. This would be a complex task to perform manually, so EDA tools have been developed that can insert TDM logic automatically, analyze timing and even choose which signals to populate the muxes with. The prime example of such an EDA tool is the Certify® tool from Synopsys, which supports a number of different asynchronous and asynchronous models for TDM.

Whichever tool is used, the problem still exists that with muxing employed, either the overall FPGA-based prototype must run at a lower speed or the on-board paths must be able to run at a higher speed. TDM ratios of 8:1 or higher are not uncommon, in those cases, a design which runs at 16MHz inside the FPGAs must be either scaled back to 2MHz (which is hardly better than an emulator!) or the external signals must be propagated between the FPGAs at 128MHz or greater, or some compromise between these two extremes. With the need for high multiplexing ratios in some FPGA-based prototypes, the overall performance can be limited by the speed at which TDM paths can be run. A good differentiator between boards is therefore their ability to run external signals at high speed and with good reliability; a noisy board will possibly introduce glitches into the TDM stream and upset the synchronizations between its ends.

Beyond simple TDM, it is becoming possible to use the LVDS (low voltage differential signaling) capability of modern FPGA pins in order to run on-board paths at speeds up to 1GHz. This full speed requires very good board-level propagation characteristics between the FPGAs. Figure 65 gives a non-detailed example of a serial TDM arrangement, which allows eight signals to be transferred across one pair of differential signals (further detail in chapter 8).

At a high transmission speed of 800MHz and a multiplexing ratio of 8:1, the 128MHz speed in our previous example could easily be supported, and even increased to a ratio of 40:1, as long as the board is good enough. The possibility to run much higher mux ratios gives a large increase in the usable connectivity between FPGAs. For example, a TDM ratio of 64:2 (2 rather than 1 because of the differential pins required) could be supported on prototypes running at 30MHz or

Figure 65: High Speed TDM using differential signaling between FPGAs



more, making it much easier for the EDA tools to partition and map any given design. There is more on the discussion of TDM in chapter 8.

So, once again, a good differentiator between FPGA boards is their ability to support good quality LVDS signaling for higher overall performance and to supply the necessary voltage and IO power to the FPGAs to support LVDS. It is also important that designers and indeed the tools they employ for partitioning understand which board traces will be matched in length and able to carry good quality differential signals. A good board will offer as many of such pairs as possible and will offer tool or utility support to allocate those pairs to the most appropriate signals. For example, the HAPS-60 series boards from Synopsys have been designed to maximize opportunity for LVDS-based TDM. Details of LVDS capability in each Virtex-6 device and signal routing on the HAPS-60 board is built into the Certify tools to provide an automated HSTDMD (high-speed TDM) capability. This example again underlines the benefit of using boards and tools developed in harness.

6.9. Cost and robustness

Appendix B details the hidden costs in making a board vs. using a commercial off-the-shelf board including discussions about risk, wastage, support,

documentation and warranty. The challenges of making our own boards will not be repeated here but it is worth reminding ourselves that these items apply to a greater or lesser degree to our choice of commercial boards too, albeit, many are outsourced to the board provider.

We should recall that total cost of ownership of a board and its initial purchase price are rather different things. Balanced against initial capital cost is the ability to reuse across multiple projects and the risk that the board will not perform as required. It may not be sensible to risk a multi-year, multi-million dollar SoC project on an apparently cheap board.

Of particular note is the robustness of the boards. It is probable that multiple prototype platforms will be built and most may be sent to other end-users, possibly in remote locations. Some FPGA-based prototyping projects even require that the board be deployed in situ in an environment similar to the final silicon e.g., in-car or in a customer or field location. It is important that the prototyping platform is both reliable and robust enough to survive the journey.

Recommendation: boards which have a proven track record of off-site deployment and reliability will pay back an investment sooner than a board that is in constant need of attention to keep it operational. The vendor should be willing to warranty their boards for immediate replacement from stock if they are found to be unreliable or unfit for use during any given warranty period.

6.9.1. Supply of FPGAs governs delivery of boards

Vendors differentiate their boards by using leading-edge technology to ensure best performance and economies of scale. However, these latest generation FPGAs are typically in short supply during their early production ramp-up, therefore, a significant commercial difference between board vendors is their ability to secure sufficient FPGA devices in order to ensure delivery to their customers. While some board vendors may have enough FPGAs to make demonstrations and low volume shipments, they must generally be a high-volume customer of the FPGA vendors themselves in order to guarantee their own device availability. This availability is particularly important if a prototyping project is mid-stream and a new need for boards is recognized, perhaps for increased software validation capability.

6.10. Capacity

It may seem obvious that the board must be big enough to take the design but there is a good deal of misunderstanding, and indeed misinformation, around board capacity. This is largely driven by various capacity claims for the FPGAs employed.

The capacity claimed by any vendor for its boards must be examined closely and questions asked regarding the reasoning behind the claim. If we blindly accept that such-and-such a board supports “20 million gates,” but then weeks after delivery find that the SoC design does not fit on the board, then we may jeopardize the whole SoC and software project.

Relative comparison between boards is fairly simple, for example, a board with four FPGAs will hold twice as much design as a board with two of the same FPGAs. However, it is when the FPGAs are different that comparison becomes more difficult, for example, how much more capacity is on a board with four Xilinx® XC6VLX760 devices compared with a board with four XC6VLX550T devices? Inspecting the Xilinx® datasheets, we see the figures shown in Table 13.

Table 13: Comparing FPGA Resources in different Virtex®-6 devices

FPGA Resource	XCE6VLX550T	XC6VLX760	Ratio 760:550
logic cells	549,888	758,784	138%
FF	687,360	948,480	138%
BlockRAM (Kbit)	22,752	25,920	114%
DSP	864	864	100%
GTX Transceivers	36	0	n/a

We can see that, in the case of logic cells, the LX760-based board has 38% more capacity than the LX550T-based board, but with regard to DSP blocks, they are the same. There is another obvious difference in that the 550T includes 36 GTX blocks but the LX760 does not have the GTX feature at all.

Which is more important for a given design, or for future expected designs? The critical resource for different designs may change; this design may need more arithmetic but the next design may demand enormous RAMs. Readers may be thinking that visibility of future designs is limited by new project teams often not knowing what’s needed beyond perhaps a 12-month window. For this reason, it is very helpful for prototypers to have a place at the table when new products are being architected in order to get as much insight into future capacity needs as possible. This is one of the procedural recommendations given in our Design-for-Prototyping manifesto in chapter 9.

Getting back to current design, we should always have performed a first-pass mapping of the SoC design into the chosen FPGA family, using the project’s

intended synthesis tool in order to get a “shopping list” of required FPGA resources (as discussed in chapter 4).

Project success may depend on other design factors but at least we will be starting with sufficient total resources on our boards. We may still be tempted to use partitioning tools and our design knowledge in order to fit the design into four FPGAs at higher than 50% utilization, but we should beware of false economies. The advice that some find difficult to accept is that economizing on board capacity at the start of a project can waste a great deal of time later in the project as a growing design struggles to fit into the available space.

Recommendation: in general, ignore the gate count claims of the board vendor and instead run FPGA synthesis to obtain results for real resource requirements. We should add a margin onto those results (a safe margin is to double the results) and then compare that to the actual resources provided by the candidate board.

6.11. Summary

Choosing FPGA boards is a critical and early decision in any FPGA-based prototyping project.

A wrong decision will add cost and risk to the entire SoC project. On the face of it, there are many vendors who seem to offer prototyping boards based on identical FPGAs. How might the user select the most appropriate board for their project? The checklist list in Table 14 summarizes points made during this chapter and can help.

Table 14: A top-10 checklist for assessing FPGA prototyping boards

1	What range of base boards is available?	Are they mutually compatible and additive?
2	What is the range of add-on boards and daughter cards?	Which IP, microprocessor, and connectivity standards?
3	How flexible is the interconnect?	Can it meet performance needs of multiple projects?
4	Does the board have proven record of quality and reliability?	Are reference customers able to verify the board vendor's claims?
5	Will the board be supported locally?	Is training available? How complete is documentation?
6	What performance has been achieved by other users of the board?	Are reference customers able to verify claims?
7	Is the board portable and robust enough to be re-usable for future projects?	Can this be demonstrated?
8	What is the on-board debug capability?	Can the board be debugged remotely?
9	What links to other debug and verification tools can be provided?	Has it been proven to work with existing verification tools?
10	Does the vendor offer an extended warranty?	Is a quick replacement option available?

It is important to remember that a prototyping project has many steps and choosing the board, while important, is only the first of them.

This chapter describes the main prototyping challenges, common practices and the process of taking an SoC design into the FPGA-based prototyping system. It covers SoC design-related issues, techniques to make the design prototyping friendly and how to use FPGA special-purpose resources. We cover SoC library cells, memories and clock gating in depth. We also revisit the implementation process and common tools outlined in chapter 3 in order to accomplish the best system performance.

7.1. Why “get the design ready to prototype?”

While the aim is always to prototype the SoC source RTL in its original form, early on in the prototyping effort it typically becomes evident that the SoC design will have to be modified for it to fit into the prototyping system. The design variations are typically due to design elements found in the SoC technology which are not available in, or suitable for FPGA technology. Design variations are also caused by limitations in the prototyping platform, tweaking for higher performance and debug instrumentation. The typical design variations, and examples of how to best handle them, are discussed in more detail later in this chapter.

To facilitate the SoC design modifications, we may make copies of affected source files, edit them and then simply replace the originals for the duration of the prototyping project. Obviously care and revision control methods should be employed to avoid error but in the end, we are probably going to be changing the RTL at some time.

7.1.1. RTL modifications for prototyping

Table 15: SoC design elements that might require RTL changes

Top-level pads	Instantiations of SoC pads will not be understood by the FPGA tool flow.
Gate-level netlists	The design is not available in RTL form, but only as a mapped netlist of SoC library cells. These will not be understood by the FPGA tool flow.
SoC cell instantiations	Leaf cells from the SoC library are instantiated into the RTL, for whatever reason, and they will also not be understood by the FPGA tool flow.
SoC memories	Instantiations of SoC memory will not be understood by the FPGA tool flow.
SoC-specific IP	From simple DesignWare macros up to full CPU sub-systems, if the source RTL for the IP is not available then we will need to insert an equivalent.
BIST	Built-in self test (BIST) and other test-related circuitry is mostly inferred during the SoC flow but some is also instantiated directly into the RTL. This is not required for the prototype and may not be understood by the tools.
Gated clocks	As with BIST, clock gating can be inferred by SoC tools but is often written directly into the RTL. This generally overflows the clock resources available in the FPGAs.
Complex generated clocks	As with gated clocks, generated clocks might require simplification or otherwise handling in order to fit into the FPGA.

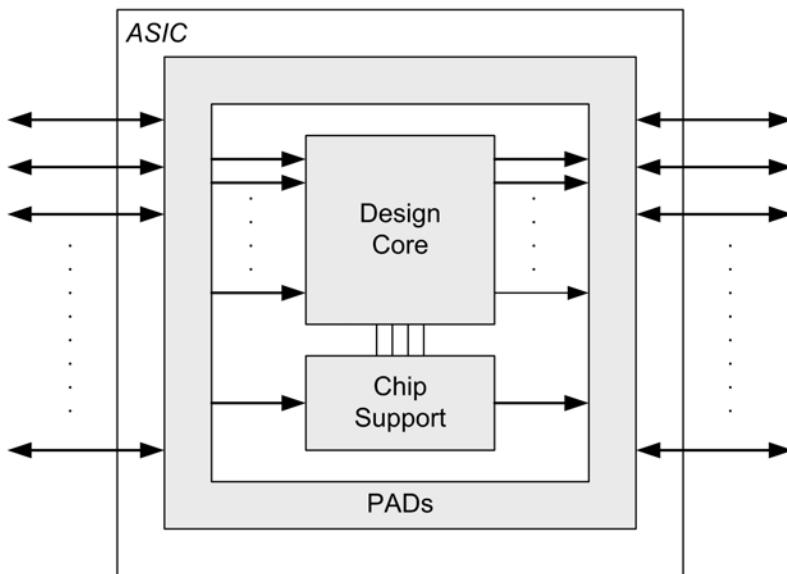
The most common elements in SoC designs that need to be modified during prototyping include those listed in Table 15.

In addition, there are other reasons why we might need to alter the original RTL, including test-point insertion, tie-off of unused inputs, instantiation of FPGA-specific clocks etc. Each of these types of RTL change will be explained in the following pages and we will give some practical ways to implement them.

7.2. Adapting the design's top level

We asked ourselves a rhetorical question in chapter 4 which was: “how much of the design should we prototype?” and it is time to answer that question in more detail. We will find a number of technology-specific elements that will not work in our FPGAs, usually at the top level of an SoC design hierarchy. These are the chip support elements and the top-level IO pad instantiations. A rough diagram of an SoC top-level is shown in Figure 66 where we see the SoC-specific chip support alongside the majority of the SoC logic in the core block, which is the top of the rest of the design hierarchy.

Figure 66: Simplified view of SoC top-level



To introduce this design into FPGA, we will need to either replace the chip support and IO pads with FPGA equivalents, or simply remove the top level entirely and wrap the design core with a new FPGA-specific top level. We will address the chip support block in a moment, but first, how do we handle the IO?

7.2.1. Handling the IO pads

FPGA synthesis does not need to have IO pad instantiations in the RTL because it is able to infer an FPGA pad and even configure it in most cases, using only the defaults or simple attributes attached to the top-level signal. We could therefore simply leave the pads out and tie the dangling connections inactive or to the top-level boundary as required.

An alternative approach is to leave the IP pad ring in place and to replace each IO pad instance with a synthesizable model of its FPGA equivalent.

A typical IO pad from a silicon technology library may have 20 or more connections at its boundary, including the main input and output plus voltage and slew controls and scan test. Some of these connections will link to the package pins/balls while others connect into the core of the design or directly to adjacent pads.

For the purposes of prototyping, we need only model the logical connection from the design core to the “outside” world. Therefore, we need only a simpler form of the pad which makes that logical connection, omitting the scan etc. We can easily make a small RTL file which fits into the IO pad instantiation in the SoC RTL but contains the FPGA subset equivalent. This converts a black box pad instantiation into something that the FPGA synthesis can use.

Although there may be over a thousand pads in the SoC, there may be only ten or so different types of pad. Replacing each type with an FPGA equivalent will be relatively simple, especially if our SoC designs use the same pad library over multiple projects and we can build up a small library of equivalents.

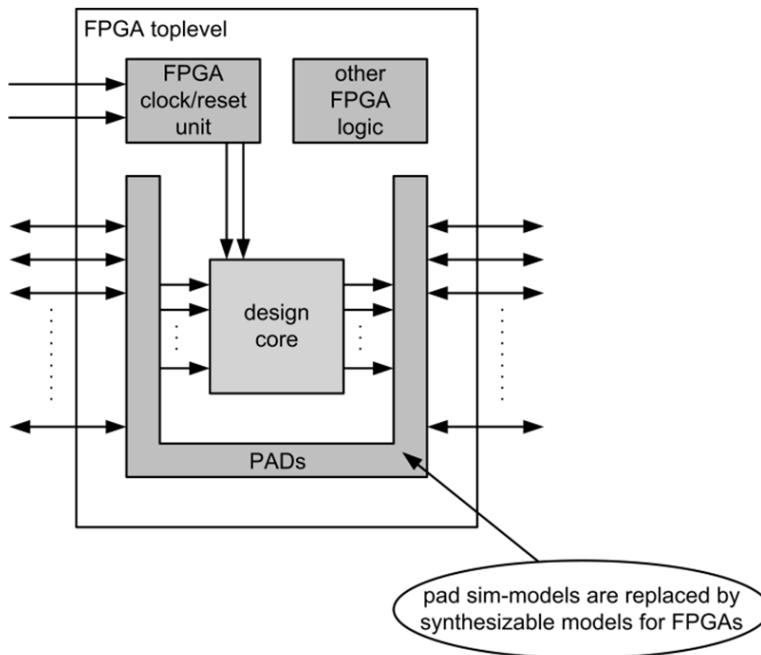
7.2.2. Handling top-level chip support elements

The block in Figure 66, labeled “Chip Support,” contains those elements in the design that are generally target specific, often seen as secondary to the main function of the RTL, and yet are essential to its correct operation. This might include such functions as clock generation and distribution, reset control and synchronization, power-gating control and test and debug control. How much of this is relevant for our prototype or is even needed for an FPGA implementation?

Some teams recommend simply replacing the chip support block with another, simpler block which takes care of those elements needed for the FPGAs. This means that, in effect we have a new FPGA-compatible version of the top-level of the SoC. The top-level RTL file for the SoC can be used as the basis for the new FPGA top-level and an example of what the new top-level might look like is shown in the block diagram in Figure 67.

Here we see the clock generation and synchronization circuits at the top level

Figure 67: New design top-level for prototype

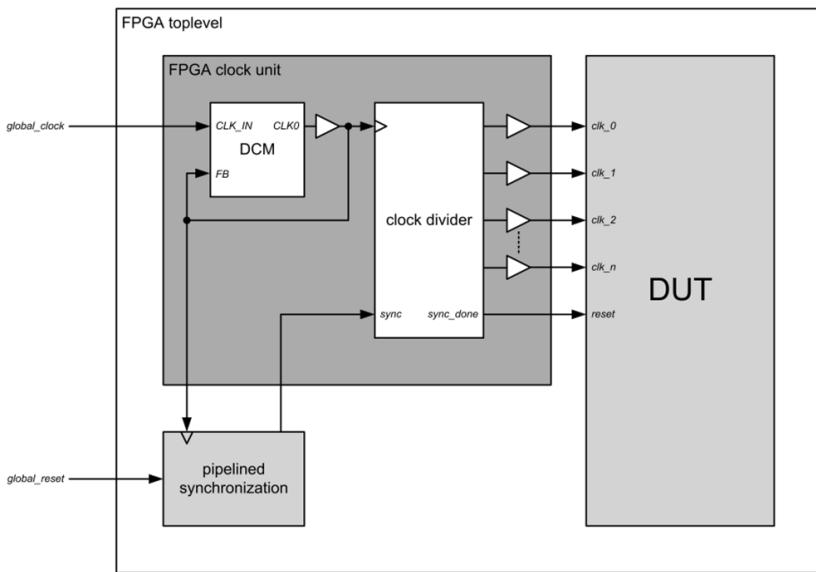


supporting the existing design core. The creation of the equivalent FPGA chip support block is a relatively simple FPGA design task involving dividers, clock buffers and synchronizers, as shown in Figure 68.

The use of the FPGA clock networks becomes more complex when the prototype uses multiple FPGAs so we shall revisit the top-level again in chapter 8, where we explore partitioned designs.

Let's look now more closely at how we handle clock gating, one of the most important tasks in making a design FPGA-ready.

Figure 68: Simple illustration of top-level chip support block for FPGA implementation



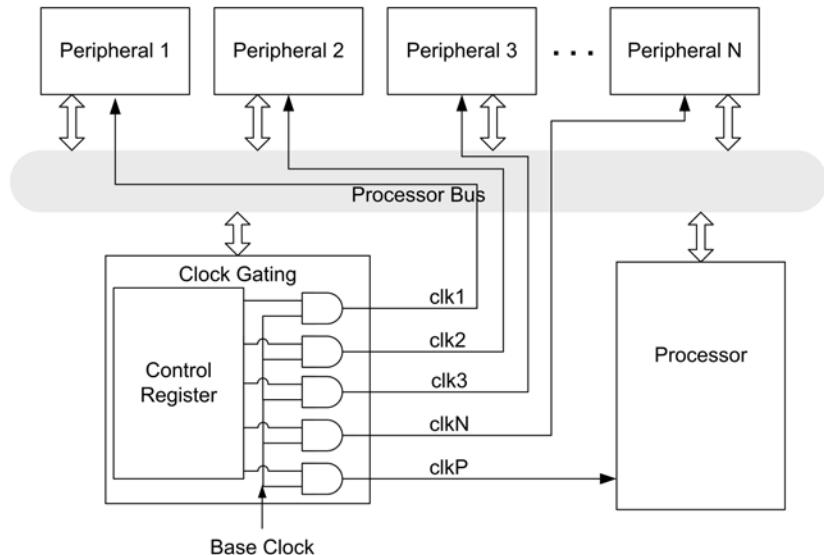
7.3. Clock gating

Clock gating is a methodology of turning off the clock for a particular block when it is not needed and is used by most SoC designs today as an effective technique to save dynamic power. In SoC designs clock gating may be done at two levels:

- **Clock RTL gating** is designed into the SoC architecture and coded as part of the RTL functionality. It stops the clocks for individual blocks when those blocks are inactive, effectively disabling all functionality of those blocks. Because large blocks of logic are not switching for many cycles it saves substantial dynamic power. The simplest and most common form of clock gating is when a logical “AND” function is used to selectively disable the clock to individual blocks by a control signal, as illustrated in Figure 69.

- During synthesis, the tools identify groups of FFs which share a common enable control signal and use them to selectively switch off the clocks to those groups of flops.

Figure 69: RTL clock gating in SoC to reduce dynamic power



Both of these clock-gating methods will eventually introduce physical gates in the clock paths which control their downstream clocks. These gates could introduce clock skew and lead to setup and hold-time violations even when mapped into the SoC, however, this is compensated for by the clock-tree synthesis and layout tools at various stages of the SoC back-end flow. Clock-tree synthesis for SoC designs balances the clock buffering, segmentation and routing between the sources and destinations to ensure timing closure, even if those paths include clock gating.

This is not possible in FPGA technology, so some other method will be required to map the SoC design if it contains a large number of gated clocks or complex clock networks.

7.3.1. Problems of clock gating in FPGA

As we saw in chapter 3, all FPGA devices have dedicated low-skew clock tree networks called global clocks. These are limited in number, but they can clock all

sequential resources in an FPGA at frequencies of many hundreds of megahertz. Owing to diligent chip design by the FPGA vendors, the clock networks also have skew of only a few tens of picoseconds between any two destinations in the FPGA. Therefore, it is always advisable to use these global clocks when we target a design into FPGAs.

However, FPGA clock resources are not suited to creating a large number of relatively small clock domains, such as we commonly find in SoCs. On the contrary, an FPGA is better suited to implementing a small number of large synchronous clock networks which can be considered global across the device.

Global clock networks are very useful, but may not be flexible enough to represent the clocking needs of a sophisticated SoC design, especially if the clock gating is performed in the RTL. This is because physical gates are introduced into the clock paths by the clock-gating procedure and the global clock lines cannot naturally accommodate these physical gates. As a consequence, the place & route tools will be forced to use other on-chip routing resources for the clock networks with inserted gates, usually resulting in large clock skews between different paths to destination registers.

A possible exception to this happens when architecture-level clock gating is employed in the SoC, for example when using coarse-grained on-off control for clocks in order to reduce dynamic power consumption. In those cases it may be possible to partition all the loads for the gated clock into the same FPGA and drive them from the same clock driver block. The clock driver blocks in the latest FPGAs, for example, the clock management tiles (CMT) in Virtex-6 devices with their mixed-mode clock managers (MMCMs) have different controls to allow control of the clock output. Some clock-domain on-off control could be modeled using this coarse-grained capability of the CMT.

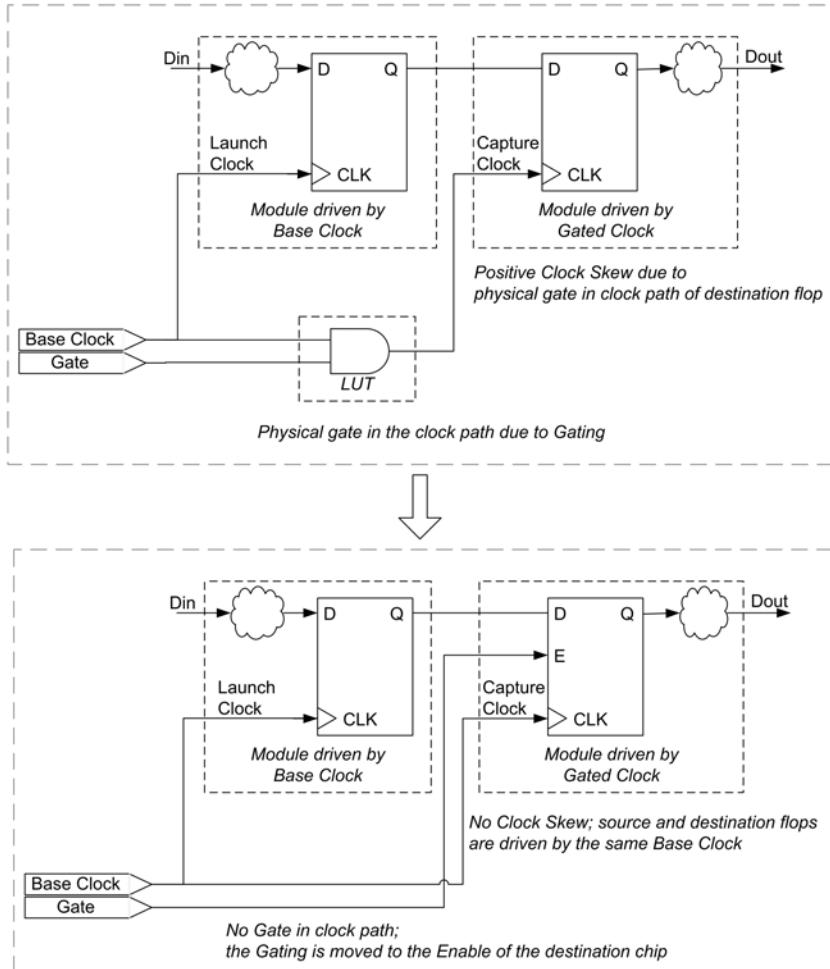
In some SoC designs there may also be paths in the design with source and destination FFs driven by different related clocks e.g., a clock and a derived gated clock created by a physical gate in the clock path, as shown in Figure 70. It is quite possible that the data from the source FF will reach the destination FF quicker/later than the gated clock, and this race condition can lead to timing violations.

7.3.2. Converting gated clocks

The solution to the above race condition is to separate the base clock and gating from the gated clock. Then route the separated base clock to the clock and gating to the clock enables of all the sequential elements. When the clock is to be switched “on,” the sequential elements will be enabled and when the clock is to be switched “off,” the sequential elements will be disabled. Typically, many gated clocks are derived from the same base clock, so separating the gating from the clock allows a single global clock line to be used for many gated clocks. This way the functionality

is preserved and logic gates present in the clock path are moved into the datapath, which eliminates the clock skew as illustrated in Figure 70.

Figure 70: Gated clock conversion and how it eliminates clock skew



This process is called gated clock conversion. All the sequential elements in an FPGA have dedicated clock-enable inputs so in most of the cases, the gated clock conversion could use this and not require any extra FPGA resource. However, manually converting gated clocks to equivalent enables is a difficult and error-prone process, although it could be made a little easier if the clock gating in the SoC design were all performed at the same place in the design hierarchy, rather than scattered throughout various sub-functions.

As we saw in Figure 66 earlier, the chip support block at the top level could include all the clock generation and clock gating necessary to drive the whole SoC. Then, during prototyping, this chip support block can be replaced with its FPGA equivalent. At the same time, we can manually replace the clock gates, either instantiated or inferred, with an enable signal which can be routed throughout the device. This would then perform the role of enabling only a single edge of the global clock at each time that the original gated clock would have risen.

In most cases, manual manipulation is not possible owing to complexity, for example, if clocks are gated locally at many different always or process blocks in the RTL. In that case, and probably as the default in most design flows, automated gated-clock conversion can be employed.

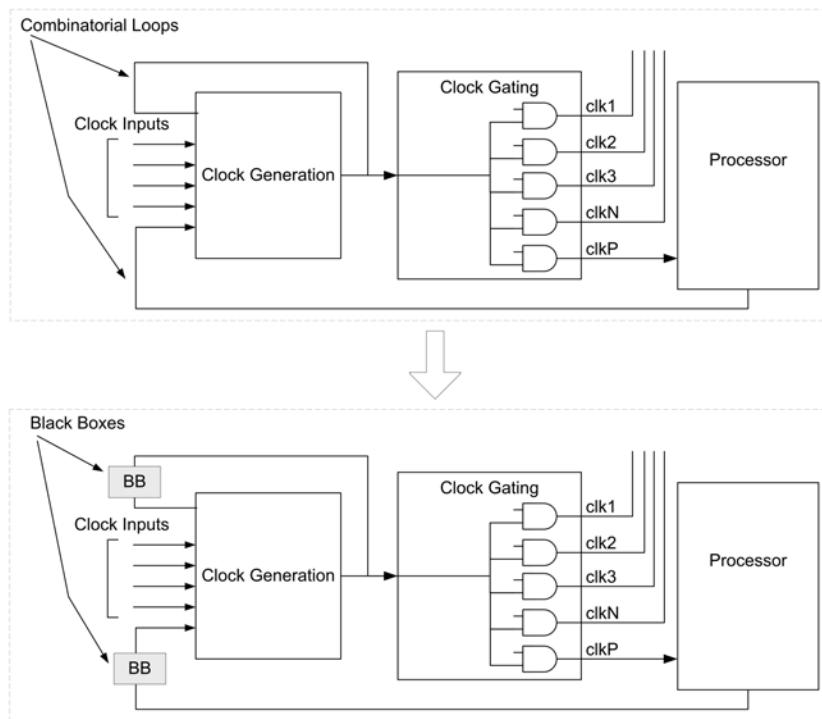
7.4. Automatic gated-clock conversion

Modern FPGA synthesis tools perform this gated-clock conversion process automatically without us having to change the RTL, however, we may need to guide the synthesis tools appropriately to perform the gated-clock conversion. It should be noted that some tools are more capable than others in this task.

Here are some of the guidelines to make the synthesis tools convert the gated clocks successfully.

- Identify the base clocks and define them to the synthesis tool by adding frequency or period constraints.
- Do not define the downstream gated clocks as clocks. Remove any period or frequency constraints on the gated clocks, which may have been specified during the SoC flow.
- Set any necessary controls in the synthesis tools to enable gated-clock conversion.
- Identify any black boxes in the design which are driven by gated clocks. To fix gated clocks that drive black boxes, the clock and clock-enable signal inputs to the black boxes must be identified. Synthesis tool specific directives should be used to identify them.
- If there are combinatorial loops in the clock-gating logic then the combinatorial loops should be broken. This can be done by inserting a feed-through black box, which is a black box with one input and one output and is placed in the combinatorial loop paths as shown in Figure 71. We can then create a separate netlist for the black box with the output simply connected to input. And the created netlist for the black box must then be added to the design during place and route.

Figure 71: Interrupting combinatorial loops to enable clock gating



When all the above guidelines are followed then the synthesis tools can automatically convert all the convertible gated clocks.

The gated clock is convertible when all of the following conditions are met.

- For certain combinations of the gating signals, the gated-clock output must be capable of being disabled.
- For the remaining combinations of the gating signals, the gated-clock output should equal either the base clock or its inverted value.
- The gated clock is derived based on only one base clock.

Figure 72: Examples of convertible and non-convertible clock gates

Convertible		<p>When the value of Gate input is 0, the Gated CLK output is disabled. When the value of Gate is 1, the Gated CLK follows the Base CLK. This satisfies all the conditions. So AND logic is convertible.</p>
Non Convertible		<p>Gated CLK output cannot be disabled for either of the values of the Gate input. This violates the first condition and hence XOR is not convertible.</p>
Non-convertible		<p>Gated CLK is derived based on two base clocks. This violates the third condition and hence MUX is not automatically convertible.</p>

In order to illustrate these guidelines, Figure 72 gives some examples of simple convertible and non-convertible gates.

FPGA synthesis tools report all the converted and non-converted sequential components in its log files. The tools also list the reasons why the conversion did not happen for the non-converted sequential components. It is always advisable to look at these reports to make sure that the gated-clock conversion had happened for all the necessary sequential components.

7.4.1. Handling non-convertible gating logic

For an SoC design to work reliably on an FPGA-based prototype, all the gated clocks in the design should be converted. If the gated clock is derived based on multiple clocks, or the gating logic is complex, then synthesis tools cannot do the gated-clock conversion. However, these scenarios are sometimes common in SoC designs which can lead to many setup and hold-time violations. Here are some of the ways in which these scenarios can be handled. Use all these methods collectively as applicable.

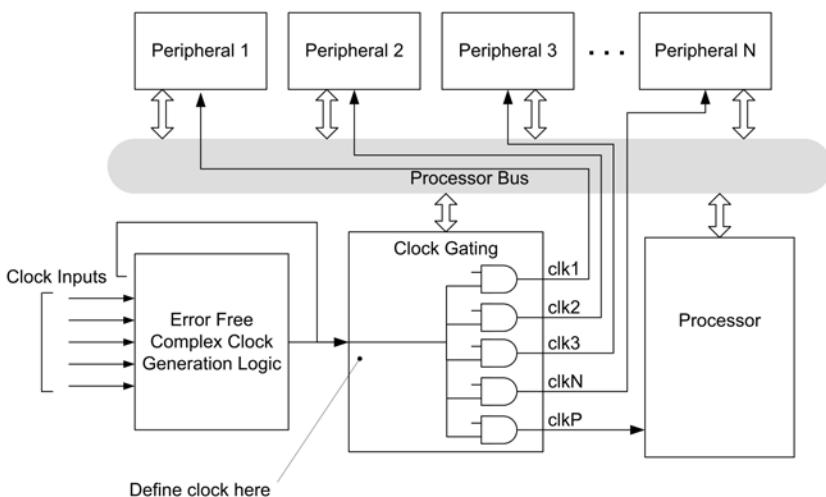
- If there are no paths between the sequential elements driven by the base clocks and the unconverted gated clock, then the latter will not create any cross-domain timing violations. However, their routing in the FPGA may need to be carefully controlled to avoid the races described above.

An intermediate node in the design can be identified and defined as a base clock such that the gating logic present, driven by that node, is convertible. Usually, SoC designs will have a clock-generation logic block with complicated logic to generate

a glitch free, fail-safe and error-free clock. This clock will be created based on switching between many different clocks. And this generated clock will be used as the base clock for the rest of the blocks in the design with individual gating logic. Defining the clock on the output of the clock-generation logic block will make sure that all the gated clocks created, based on this clock, will be converted by the synthesis tool as shown in

- Figure 73.

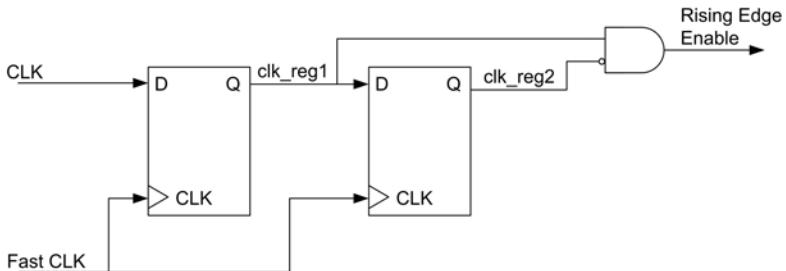
Figure 73: Handling complex clock gating



- If there are valid timing paths between one base clock and its complex gated clock, then try to manually balance the clock paths between these paths. It can be balanced by introducing feed-through LUTs, clock buffers, PLLs and digital clock managers in one of the clock paths.

- If there are still some gated clocks which are not converted, and there are huge valid timing violations, then try to run all the sequential elements in the FPGA at very high frequency – around 10x the fastest clock in the design. Insert rising-edge detectors with respect to the faster clock for all the gated clocks in the design. This rising-edge detector can be designed by double registering (say `clk_reg1` and `clk_reg2`) the gated-clock signals using the faster clock and then forming a logic to detect the change from LOW to HIGH ($\text{NOT}(\text{clk_reg2}) \text{ AND } \text{clk_reg1}$) as shown in
- Figure 74. If the original clock drives FFs which operate on a negative edge also, then negative-edge detector circuits will also be required.

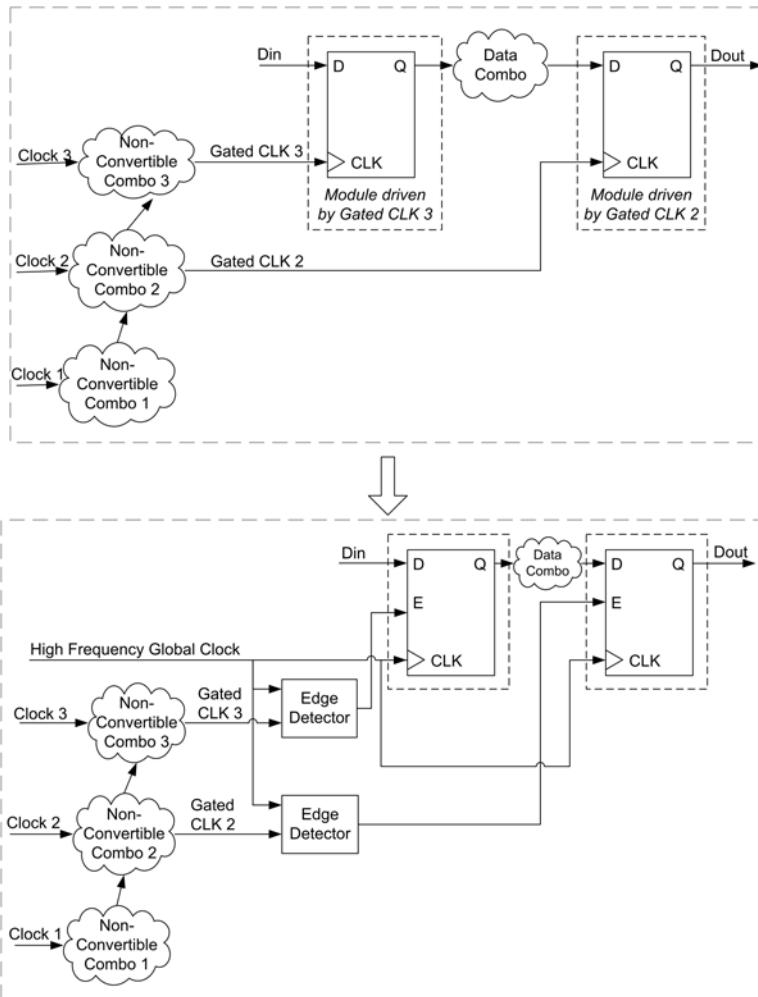
Figure 74: Rising edge detector



care must be taken in the layout of these edge detectors during place and route to avoid introducing differential delay between the paths `clk_reg1` and `clk_reg2`. Use the outputs of these edge detectors as enables on all the sequential elements which were originally driven by the corresponding gated/generated clocks.

In this way, the whole of the FPGA is driven by a single faster clock source as shown in Figure 75. This clock will use the dedicated global routing resources in the FPGA and therefore the associated clock skew will be very minimal and the timing can be easily met.

Figure 75: Handling complex clock gating with global clock



7.4.2. Clock gating summary

Clock gating is common in SoC designs and gated clocks should be handled with care to successfully prototype the SoC designs on FPGA. Contemporary FPGA synthesis tools automatically take care of most of these gated clocks when properly constrained. By following the guidelines in this chapter, SoC designs with complex clock gating can also be handled and successfully prototyped in FPGAs.

7.5. Selecting a subset of the design for prototyping

Most SoC designs include SoC technology elements that are not available in FPGA technology such as PLL, analog circuitry, BIST, SoC primitives and third-party IP. The following paragraphs describe some options to deal with SoC design elements that do not map into FPGA.

7.5.1. SoC block removal and its effect

In the cases where SoC design elements are not available in FPGA technology, or where there is no desire to prototype certain blocks, these blocks need to be removed from the design. The removal may be as simple as removing a complete RTL design file from the project, leaving an unpaired module definition which might be inferred automatically as a black box. Alternatively, the RTL design file may need to be replaced with a dummy file which explicitly includes the necessary directive to set the module as a black box for synthesis.

In less tidy arrangements, the element to be removed may be buried in an RTL file alongside other logic that we wish to keep. In that case it may be necessary to alter the RTL, but a better approach, as we shall see in chapter 9, would be to predict at the time that the RTL was written that the element might need to be removed.

Figure 76: removing a block from the design using `ifdef

```
/* defining the compiler macro for prototyping only*/
`define FPGA;

<other rtl code>

/* soc_block is not instantiated when FPGA macro is defined
(leaving the commented instantiation line helps others to
understand what has been done. Proper documentation would also be
helpful ) */

`ifdef FPGA
// soc_block block1 (signals...);

/* instantiating soc_block only when FPGA macro is not defined*/

`else
soc_block block1 (signals...);
`endif

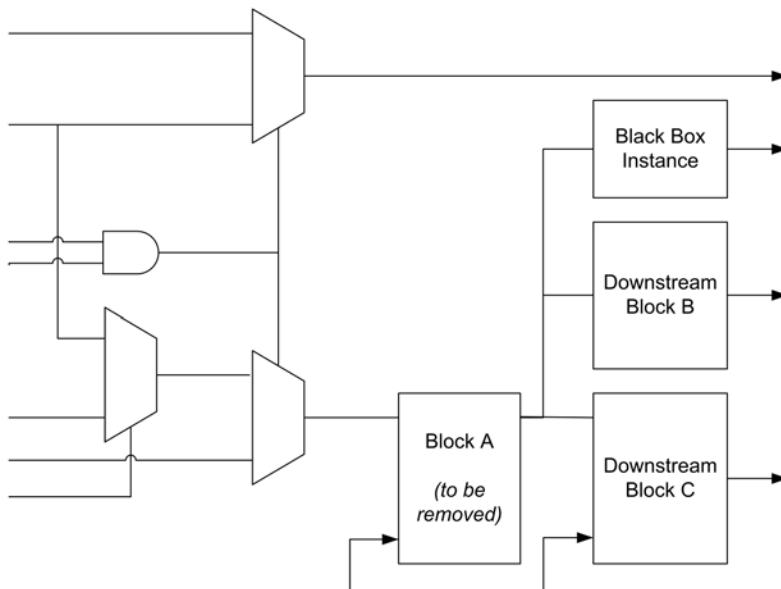
<other rtl code>
```

Taking this approach, a conditional branch may have been placed into the original RTL code, based on a single macro. The example in Figure 76 shows the condition removal of a module using `define and `ifdef.

The rest of the design that would normally connect with the removed logic may be handled in different ways by different synthesis tools. Some tools will simply flag the condition as an error error-out because there are dangling connections, but other tools will also remove downstream logic which is not driven as a result of the change; either up to a block boundary or other synthesis-invariant point, up to and including the entire cone of downstream logic. Upstream, that logic which previously drove the removed block will also be pruned as far as any upstream sources which also drive other, non-pruned, cones of logic.

To illustrate this effect, let us consider the small excerpt from a design shown in Figure 77.

Figure 77: Typical excerpt of logic before block removal

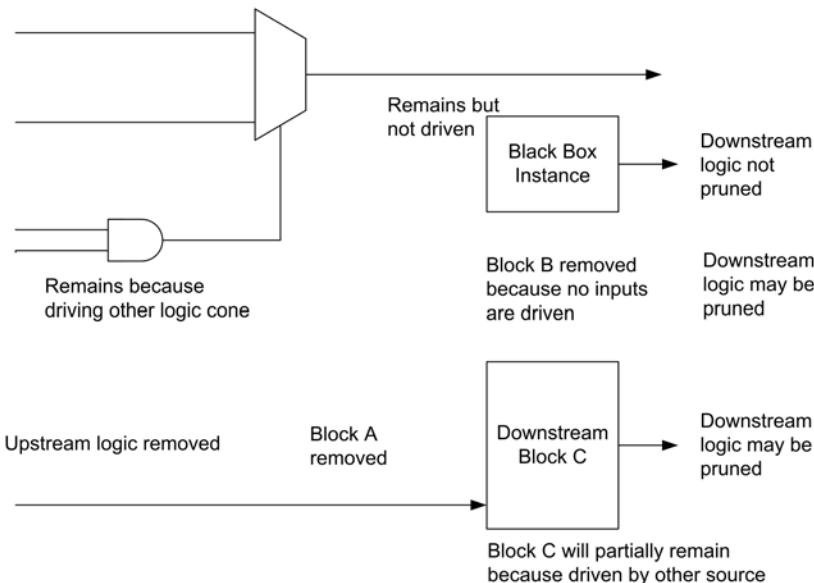


Let us suppose that Block A is not required in the prototype so we intend to simply remove it and allow the synthesis to prune out the unnecessary logic. The ripple-out effect of Block A's removal upstream and downstream will depend upon the synthesis tool's defaults and configuration. In Figure 78 we see that for a specific synthesis tool set-up, the upstream and downstream logic is mostly pruned as we might expect. Block A is removed along with all mutually exclusive upstream logic.

The AND gate is the point at which pruning stops because it is also driving other logic.

Downstream, two effects are instructive. Firstly, the downstream blocks receiving inputs from the removed block will be removed in turn unless they also receive data inputs from other parts of the design (clock or reset inputs are not sufficient). So in our example, Block B is completely removed, probably causing further pruning of ITS downstream logic. Block C will undergo pruning because some internal logic will not be driven when Block A is removed, while some other logic will remain because it is driven from other sources.

Figure 78: Effect of removing Block A from Figure 77 and resultant pruning



The second item of note is that the black box may not be removed by default and neither will its downstream logic, with the consequence that back-end tools will need to handle the lack of inputs, which may or may not be possible. This can manifest itself in some very subtle ways, for example, an instantiated BlockRAM may survive until place and route and then prevent completion because of non-driven inputs. Even worse may be the setting of non-driven inputs to unexpected values and the BlockRAM remains in circuit.

Thus, we should take care with simple removal of blocks as a method for reducing the size of our prototype. Trimming of upstream logic which is unique to the removed block is not usually a problem; however, pruning of downstream logic may have widespread and unpredictable effects.

Synthesis tool settings for cross-boundary optimizations will have an effect to promote or prevent the ripple-out of logic removal. There may also be tool-specific directives and attributes within the design or project files or the tool itself which control logic pruning. Readers are encouraged to explore their own FPGA synthesis tools in order to understand their behavior in these circumstances. For the Synopsys® example, Synplify Pro® synthesis tools have a directive called syn_noprune which, as the name suggests, prevents pruning of non-terminated or non-driven signals.

However, even if such a directive is used in synthesis, it may be that the place & route tools will have their own default operation which overrides the setting when dangling signals are found in the input netlist. It is good practice to ensure that every system or toolset used for the project is explicitly given a predictable setup rather than rely on defaults. If the members of our SoC and/or prototyping project teams are running different tools installations and seeing different results in their respective labs, then tool defaults and settings are a good place to start looking for the reasons.

So, block removal is a quick and powerful way to remove unwanted logic but may have unforeseen results. A better approach may be to replace the block with a simple set of constant values called stubs, which leave no room for ambiguity of tool dependence. There is more detail on the use of stubs in the next section.

7.5.2. SoC element tie-off with stubs

To remove possible ambiguity between tool-flows as a result of element removal, as discussed above, a simple step is to add a dummy design file which explicitly ties-off unused ports to desired values. Amending the example in section 7.5.1 aboveabove, we arrive at the following code:

We see that the code is altered to call a different version of the SoC block that included the stubs which tie off outputs to specific values, controlling downstream pruning. Inputs to the stub block can be ignored with the upstream pruning taking place as normal, alternatively, dummy registers or other logic might be used to sink the input signals or bring to an external test point using global signal (see next section). Another advantage of using stubs is that it more fully defines the design for simulation purposes. Simulators have a much more particular requirement for complete RTL code and will often error-out on signals etc. by default.

Figure 79: Substituting stub block using `ifdef

```
/* defining the compiler macro */
`define FPGA;

// define stub block (preferably in separate file used only for
prototype)

module SoC_block1_stub (signals... )
  input ...;
  output ...;

// elsewhere in the SoC design
<other rtl code>

/* replacing the SoC_block instance with stub equivalent */
`ifdef FPGA
  SoC_block SoC_block1_stub (signals...);

/* instantiating SoC module SoC_block when FPGA macro is not
defined*/
`else
  SoC_block SoC_block1 (signals...);
`endif

<other rtl code>
```

Therefore, stubs are a useful way to ensure repeatable pruning results, regardless of which tool we use or its setup.

7.5.3. Minimizing and localizing RTL changes

In previous examples, `define and `ifdef are used to control the synthesis branching between the FPGA and SoC code at compile time. In some cases, original RTL designers prefer to keep their code free of implementation-specific alterations. This fits within their team's style guidance of separation and modularity. It is seen to be "cleaner" to keep as much target-specific code out of as much of the RTL as possible. In general, this also leads to a more adaptable design. There are a couple additional options that avoid unwanted ifdef's but still improve the design's adaptability.

As suggested in a few of the examples, one option is to create "libraries" for each target, for example, by isolating a sub-list of the source files that is target-specific and only including the relevant sub-list for a given target. The good points of this

approach are that it allows us to quickly compare the two lists and discern where there are differences between the two databases. The strong disadvantage is that there are still two databases, and notwithstanding the naturally higher level of separation as compared to some of the other common approaches, maintaining multiple databases means more work and frequently leads to negligence of the secondary database.

This disadvantage is minimized if the target-specific code is kept as isolated as possible from the remaining code. For example, if several SoC library primitives are instantiated directly in a large module, two copies of the large module would have to be maintained. On the other hand, if the instantiation is of the library component (containing in one case the SoC primitive and in another case the synthesizable, behavioral equivalent), there is more chance that the code can change locally, without having to implement the change in multiple files.

Figure 80: Use of VHDL Global Signal to extract signal to a test point

```
# Globals Package
package globals_pkg is
signal observe_1 : std_logic;
end globals_pkg;

# Low Level Arch
use WORK.globals_pkg.all;
architecture bhv of deep_down is
  signal what_to_watch : std_logic;
  .
  .
  Begin
    observe_1 <= what_to_watch;
  End Bhv;

# Top Level Arch
use WORK.globals_pkg.all;
architecture Bhv of top_design is
  .
  .
  Begin
    trace_pin <= observe_1;
  End;
```

Some users find that the use of XMRs (cross-module references) in Verilog or Global signals, the VHDL equivalent, helps to decrease the scope of RTL changes. In Figure 80 we see an example of bringing an internal node out to a test pin using a VHDL Global signal. In this way we need change only three files and the only boundary change is at the top-level block in order to add the test pin itself.

Another good use of XMRs is to inject signals into a lower level. For example, one of the common needs in making a design FPGA-ready is to simplify clocking. To

make this easier, SoC RTL writers are asked to keep clock generators, gates etc. in a common block at the top level. Their reason for not doing so is often that it would complicate the hierarchy boundaries to push the clocks down to the low-level modules where the clocks are used. XMRs can overcome that objection. XMRs may also be used retrospectively by the prototyping team to achieve much the same goal.

7.5.3.1. Note: netlist editing instead of RTL changes

Another approach is to use a netlist editing utility which may accompany some synthesis or partitioning tools. We explicitly specify the differences between the original database and any modifications required for the prototype directly in the synthesis netlist. The netlist editor would be run after each synthesis run, usually from a common script. This approach can be thought of as a series of overlays on the original RTL with the target-specific library compiled in parallel with the design and then stitched into the design using the netlist editor. This has all the advantages of the multiple filelist approach, maintains and utilizes the original golden SoC RTL without changes, and eliminates most issues associated with keeping RTL databases in sync.

Other approaches include using code generators and project generators. These are frequently written in Perl, C, etc., or using makefiles, or combinations thereof. In practice, most projects use combinations of approaches. For example, a project could use ifdef's for the library selection, or include them in the library itself.

However it is done, the target-specific code should be architected so that it is tightly bound to local sections of the design. Keeping the effect close to the cause avoids confusion. The best approach is the one that minimizes the impact on the design and maximizes the ease and simplicity of switching between targets.

7.5.4. SoC element replacement with equivalent RTL

In the cases where a non-synthesizable SoC design element is needed in the FPGA-based prototyping effort, it may be replaced with synthesizable RTL code. In Figure 81 we see an example of an SoC primitive SoC_mux being replaced with RTL code:

It is obviously important to verify that the RTL is functionally equivalent to the SoC module it's replacing. Therefore some simulation of the replacement alongside the original, perhaps with assertions checking for differences, would be very useful. We discuss such an approach with respect to memories in section 7.7.3 below.

Figure 81: Replacing instantiation with behavioral code

```
/* defining the compiler macro */
`define FPGA true;

module SoC_top (...inputs, outputs...);
    input ....;
    output ....;

//some internal signals definitions
wire sel;
wire in1;
wire in2;
wire mux_out;

/* replacing the SoC_mux instance with RTL */
`ifdef FPGA
    assign mux_out = sel ? in1 : in2;

/* instantiating SoC module SoC_mux when FPGA macro is not
defined*/
`else
SoC_mux SoC_mux1 (
    .input1(in1),
    .input1(in1),
    .select(sel),
    .output(mux_out)
);

`endif
endmodule
```

Recommendation: It's important to note that if the remaining design is not properly terminated after removal of some blocks then the synthesis tool may optimize out (i.e., remove) any logic that is not driven by, or is not driving, other logic as a result. Therefore additional design modifications or synthesis directives may be necessary, such as creating stub designs.

In many cases the SoC element which requires replacement is an instantiated leaf cell from the technology library. A good source of equivalent functionality for such an instantiated SoC element can be found in the actual technology library used for the SoC.

The liberty (.lib) format for technology libraries includes an equation for the functionality for each leaf cell. We can see such a function for a basic cell in the small excerpt from a .lib file shown in

Figure 82 and we can see the function of the output pin Y in logical terms of the inputs. It is a simple matter to convert this “function” equation into equivalent RTL, although there are EDA utilities which can perform the same task quickly and without error.

As an example, FPGA synthesis tools from Synopsys have the ability to read the .lib

Figure 82: Excerpt from an AND-OR cell description in .lib file

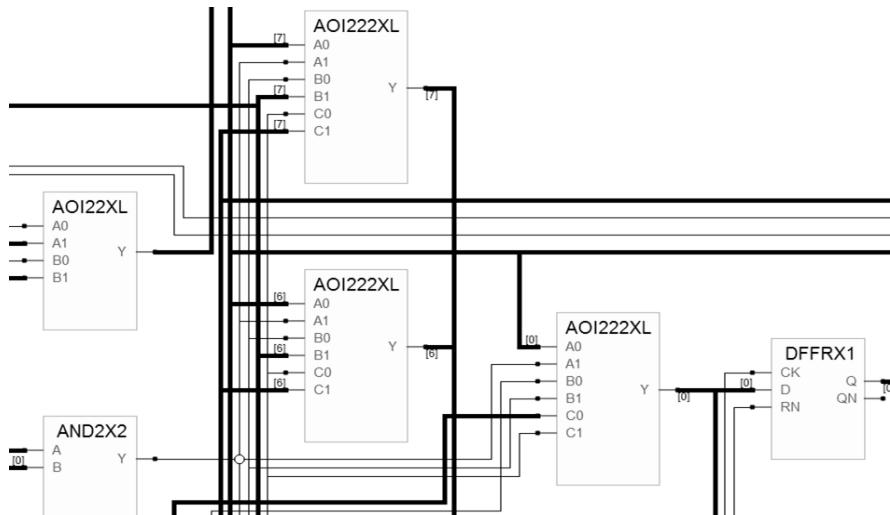
```
cell (AOI22XL) {
    cell_footprint : aoi22;
    area : 10.1844;
    pin(A0) {
        direction : input;
    }
    pin(A1) {
        direction : input;
    }
    pin(B0) {
        direction : input;
    }
    pin(B1) {
        direction : input;
    }
    pin(Y) {
        direction : output;
        function : "(!((A0 A1) | (B0 B1)))";
```

file directly and refer to the cell description in order to resolve instantiations of leaf cells in the RTL.

Figure 83 shows part of an RTL view of a gate-level netlist after being compiled into FPGA synthesis. Note that the blocks are leaf-cell elements from the technology library used for the SoC synthesis.

As noted previously, each of these cells would normally be interpreted by FPGA synthesis as a black box. Luckily, the source .lib file for the technology is available to the prototyping team and we can add this to our synthesis project as any other design file. The tools then automatically extract the functionality as mentioned, mapping it to FPGA resources during synthesis as normal.

Figure 83: Excerpt of graphical view of SoC gate-level netlist



If the .lib file is not available, or we are not using a synthesis tool that supports this flow, then a colleague supporting the SoC tool flow might be able to create a Verilog file from the .lib to add into the FPGA project. That Verilog file would only be a set of module definitions, which could be created using a utility such as Synopsys lib2syn, or even by a script which extracts the function from the .lib file and transcribes it into a Verilog module declaration.

7.5.5. SoC element replacement by inference

In some cases SoC elements can be substituted by equivalent FPGA design elements or “cores.” These FPGA cores are special-purpose FPGA entities used to optimize FPGA implementation for area and or performance. Common examples of FPGA primitives are memory blocks or shift register functions.

Usually, we do not need to simply replace an SoC element with an FPGA equivalent but instead we replace it with an RTL description and allow the synthesis tools to map it into FPGA elements by inference. The example in

Figure 84 shows an 8x4 synchronous RAM module `soc_ram` being replaced by RTL code. In this case, the FPGA memory block will be inferred by the synthesis tool during synthesis.

The RTL enabled by the FPGA macro will be interpreted by the FPGA synthesis as RAM and mapped into the relevant resource in the FPGA(s).

Figure 84: Instantiated RAM replaced by inferred equivalent

```
/* defining the compiler macro */
`define FPGA true;

module soc_top (...inputs, outputs...);
input ....;
output ....;

//some internal signals definitions
wire [3:0] mem_out;
wire [3:0] mem_in;
wire [2:0] addr;
wire we;
wire clk;

/* replace soc_ram instance with RTL when compiling for FPGA */
`ifdef FPGA

reg [3:0] mem [7:0];
assign mem_out = mem[addr];
always @(posedge clk)
if(we) mem[addr]= mem_in;

/* instantiating SoC module soc_ram when compiling for SoC */
`else
soc_ram_soc_ram1 (
.mem_do(mem_out),
.mem_di(mem_in),
.mem_addr(addr),
.mem_clk(clk),
.mem_we(we)
);
`endif
endmodule
```

7.5.6. SoC element replacement by instantiation

In addition to memory blocks, FPGAs have some special-purpose blocks that may be needed for the prototyping effort. Examples of such blocks are high-speed serial interface blocks (also known as SERDES), DDR memory interfaces and FIFOs. Since these special-purpose blocks are highly programmable, they are not always inferred by the synthesis tools and instead they must be instantiated directly into the design.

The process of including such elements is as follows:

- Create an FPGA element using the FPGA vendor's supplied tools (such as the Xilinx® CORE Generator™ tool, Memory Interface Generator etc.) Typically the user selects the core type, the target technology and defines the various parameters' initial state values etc.
- The FPGA tool generates the desired core's FPGA netlist and initialization file –where applicable – that are used in the place & route stage, and a template file used to instantiate the generated core into the main design.
- In addition, the tool generates a wrapper file containing functional simulation customization data that, combined with the primitive model used in the core, can be used for functional simulation.
- We then instantiate the template file in the design and connects the module to the design.

Figure 85: Instantiation template created by Xilinx® CORE Generator tool

```
/----- Begin Cut here for INSTANTIATION Template ---//  
// INST_TAG  
fifo_generator_v5_1 YourInstanceName (  
    .clk(clk),  
    .din(din), // Bus [17 : 0]  
    .rd_en(rd_en),  
    .rst(rst),  
    .wr_en(wr_en),  
    .dout(dout), // Bus [17 : 0]  
    .empty(empty),  
    .full(full));  
  
// INST_TAG_END ----- End INSTANTIATION Template -----
```

Copyright © 2011 Xilinx, Inc.

Figure 85 shows an instantiation of a FIFO template generated by the Xilinx® CORE Generator tool:

Figure 86: use of FIFO module in defined using the template in Figure 85

```
/* instantiating the above FIFO module in the design "top";  
  
module top (input... outputs...);  
  
/* defining the compiler macro */  
'define FPGA true;  
  
module top (input... outputs...);  
.  
<other RTL code>  
:  
'ifdef FPGA  
/* instantiation of FPGA FIFO */  
fifo_generator_v5_1 MyFIFO(  
    .clk(Myclk),  
    .din(Mydin),  
    .rd_en(Myrd_en),  
    .rst(Myrst),  
    .wr_en(Mywr_en),  
    .dout(Mydout),  
    .empty(Myempty),  
    .full(Myfull));  
  
'else  
//original SoC primitive instance  
  
soc_FIFO MyFIFO (  
    .clk(Myclk),  
    .soc_din(Mydin),  
    .soc_rd_en(Myrd_en),  
    .soc_rst(Myrst),  
    .soc_wr_en(Mywr_en),  
    .soc_dout(Mydout),  
    .soc_empty(Myempty),  
    .soc_full(Myfull));  
'endif  
  
endmodule
```

The FIFO template would be instantiated in place of SoC FIFO module as shown in Figure 86, once again using the FPGA macro to branch between implementations.

In the example, the synthesis tool will generate a netlist with an FPGA equivalent black box in place of the SoC black box. The contents are added in when the design

reaches the back-end and the association is made by the model name in the template.

When instantiating a core for which an FPGA netlist exists, the synthesis tool usually applies timing constraints to the core. Furthermore, depending on the synthesis tool, it may also be possible to include the FPGA netlist during synthesis so that further optimization may occur. In this case, we refer to the module as a gray box.

7.5.7. Controlling inference using directives

There may be situations where within the same design the use of special-purpose FPGA resources is desired for some instances, but not desired for other instances. Such situations can happen due to the finite number of available resources or the locations of these resources and the way they are connected into the design. For example, in some Xilinx® FPGA families, dedicated 48-bit multiplier blocks are available in fixed columns on the die. By default, the FPGA synthesis should map to dedicated resources but we also need to be able to override the default decisions in some situations. For example, when routing delays to a multiplier from the rest of the design placement would outweigh the performance gain in using it.

Figure 87: RTL excerpt showing control of DSP48 mapping using synthesis directive

```
<other RTL statements>
. . .
Reg [7:0] x1_in1, x1_in2; //inputs to multiplier 1
Reg [7:0] x2_in1, x2_in2; //inputs to multiplier 2

wire [15:0] mult_out1 /* synthesis syn_DSPstyle = "dsp48" */;
wire [15:0] mult_out2 /* synthesis syn_DSPstyle = "logic" */;

/* a DSP48 will be inferred for the multiplier driving mult_out1
*/
/* a DSP48 will not be inferred for the multiplier driving
mult_out2 */

Assign mult_out1 <= x1_in1 * x1_in2;
Assign mult_out2 <= x2_in1 * x2_in2;
```

In that case it would be better instead to implement the multiply function in general-purpose distributed logic. So there is a need to selectively direct the synthesis tool to infer the dedicated multipliers for some multiply functions and to

map to logic in other cases. A synthesis attribute should be available to control these kinds of decisions and override the default.

In the RTL in Figure 87 we see the use in Synopsys FPGA synthesis of an attribute called `syn_DSPstyle`. This attribute can take one of two values: “logic” or “dsp48” and it is used to direct the synthesis to infer or not to infer the 48-bit fixed multiplier.

Note that this attribute also applies to other entities that can be mapped into the DSP48 block such as adders and registers.

In some cases, this process can be done automatically when synthesis infers the use of DSP and RAM blocks. The inference is timing-driven and often paths are retimed to get better DSP and RAM packing, but also a running count of DSP and RAM usage is maintained so that if the resource limit for the target FPGA is overflowed. Then some of the design that might otherwise infer DSP and RAM blocks will be automatically mapped into other logic resources instead.

7.6. Handling RAMs

Memory is the most common SoC element that requires some manipulation to be FPGA-ready. We will focus now on RAM in particular and consider other types of memory later.

We have seen in section 7.5.5 that RAM can be described behaviorally in RTL and then inferred by FPGA synthesis into the correct FPGA memory elements. Unfortunately, SoC synthesis tools do not handle memory in the same way, and instead they are instantiated as black boxes in the RTL. Various views, such as functional behavior and physical layout, are used to “fill” the black box later in the SoC verification and implementation flows.

These various views of the memory are often automatically created and parameterized by a generator such as the coreConsultant from Synopsys or Custom Touch Memory Compilers from Virage Logic. The SoC team will be familiar with these types of tools and use them to generate very sophisticated memories, optimized for the SoC design. In an ideal world, the FPGA synthesis would recognize the black box as the output from a particular memory generator and automatically replace it with an equivalent FPGA view. However, there are a large number of possible memory configurations, as can be seen in Table 16, and the synthesis would need to infer the functionality of each of them with only the reference of the black box name from external library as a guide.

Table 16: Examples of the wide range of RAMs in use in SoC designs

RAM types	Configuration	Read/Write port details
Synchronous SRAM	Single-port	1RW
	Dual-port	2RW
Synchronous register file SRAM	One-port	1RW
	Two-port	1R,1W
	Multi-port	nR,mW
Asynchronous SRAM	Single-port	1RW
	Dual-port	2RW

Maintaining a cross-reference of all possible SoC memories from all possible generators to their closest FPGA equivalent would not be productive and typically yields non-optimal results. Instead we focus on the specific RAMs in the SoC (which will be a small subset of the overall range of possible configurations) and create optimized replacements only for them.

Some help can be offered by the memory generator tools and memory IP developers themselves and some do indeed generate FPGA equivalent views for use by prototypers. In an ideal world our SoC team will have chosen their RAM for exactly that reason, but in most cases we need to consider how we can replace an SoC RAM with the FPGA equivalent.

7.7. Handling instantiated SoC RAM in FPGA

FPGA tools don't understand any instantiated RAMs used in the SoC design. In addition, there are limits imposed by the memory architectures available in the FPGA device itself, so there is no guarantee that all types of SoC RAMs can be directly mapped into BlockRAMs or distributed RAMs that are supported in the FPGA device. Before exploring that further, here is a quick recap on FPGA RAMs (more details are in chapter 3).

7.7.1. Note: RAMs in Virtex®-6 FPGAs

In FPGAs, there are two different groups of RAMS; Block RAMs and distributed RAMs.

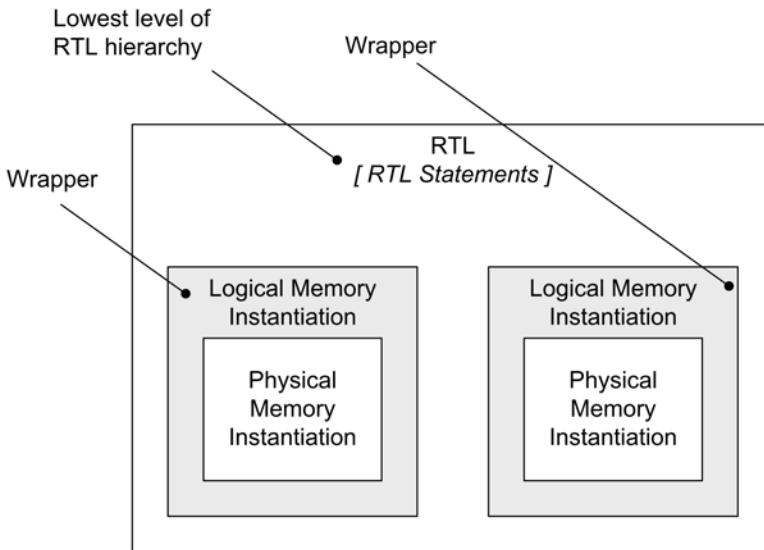
- **BlockRAM:** The Virtex®-6 block RAM stores up to 36K bits of data and can be configured as either two independent 18-Kbit RAMs, or one 36-Kbit RAM. Each 36-Kbit BlockRAM can be configured in a number of ways e.g., 32K x 1, 16K x 2 etc. they can also be cascaded to create a 64Kx1 RAM. Each 18Kb BlockRAM can be configured as a 16K x 1, 8K x2, 4K x 4, 2K x 9, or 1K x 18 memory.
- Write and read are synchronous operations. The two ports are symmetrical and totally independent, sharing only the stored data. Each port can be configured in one of the available widths, independent of the other port. The memory content can be initialized or cleared by the configuration bitstream so that contents are already present when the device comes out of reset. During a write operation the memory can be set to have the data output either remain unchanged, reflect the new data being written or the previous data now being overwritten.
- The BlockRAMs may be configured as single-port, simple dual-port and true dual-port. Furthermore, embedded dual-port or single-port RAM modules, ROM modules, synchronous FIFOs, and data width converters are easily implemented from BlockRAMs using the Xilinx® CORE Generator™ module generator tool.
- **Distributed RAM:** The look-up tables in the FPGA can also be configured as RAM and because these are spread throughout the device, we call them distributed RAMs. With the use of surrounding logic, there is great flexibility in how distributed RAM can be used.
- In cases where the SoC RAM topology is not compatible with BlockRAM (e.g., quad-port or bit-addressable RAM) then distributed RAMs can be used. In any case, if the size of the RAM is small (e.g., 16x8, 36x2) then distributed RAMs are generally a better choice than BlockRAM.
- **Interfaces to external RAM:** FPGAs have a growing amount of internal RAM but it is inefficient to use up whole FPGAs just to map a few Megabytes of RAM from an ASIC. Furthermore, most SoC designs contain much larger RAMs than a few Megabytes, so for these reasons it will probably be necessary to map some of the SoC memory to external RAM components on the boards. FPGAs have the ability to interface through fast IO to external memories and even built-in support for interfacing to standard external RAMs, such as DDR2 and DDR3.

When mapping SoC RAMs it is necessary to adapt the RTL so that the FPGA tool flow can map it into the appropriate resource. We can do this without changing the existing RTL, but instead we add extra RTL files to act as an adaptor between the black-box RAM instantiations in the SoC RTL and the necessary FPGA or external equivalent. We call these adapters “wrappers” and we shall spend some time exploring their use next.

7.7.2. Using memory wrappers

A wrapper is a small piece of RTL that contains an item to be implemented in the FPGA, but which has a top-level boundary that maps to the component/module instantiation in the SoC RTL. Experienced prototypers will be very familiar with wrappers and may indeed have built up their own libraries of wrappers for use in various situations.

Figure 88: Basic concept of a wrapper for memory



The diagram in Figure 88 shows the basic arrangement, in this case two wrappers used in the same level of hierarchy. Good practice in RTL would suggest that this would be at the lowest level of hierarchy of the SoC design but in the prototype, a wrapper adds levels below the logic already in place in the SoC. Strictly speaking this may break the style guide for the SoC project as a whole but may be preferable to editing the RTL *in situ* to add the new RAMs.

The simplest way to start creating a wrapper is to copy the component/module declaration from the SoC RTL and paste into a new RTL file. We shall later see what other items we might put in the wrapper body.

The first aim of a wrapper is to link the ports on the SoC RTL instantiation to the relevant ports on a module/component which the FPGA synthesis will understand as FPGA or external elements. This module/component may be a different black-box instantiation, for example a Xilinx® RAM macro or an external memory black box,

or it may be another layer of hierarchy in which some new RTL infers an FPGA RAM.

7.7.2.1. Wrappers to instantiate equivalent FPGA RAMs

Figure 89 shows a schematic generated as an “RTL view” by Synopsys FPGA synthesis.

Figure 89: Typical wrapper for instantiate FPGA RAM as seen in Synplify® RTL View

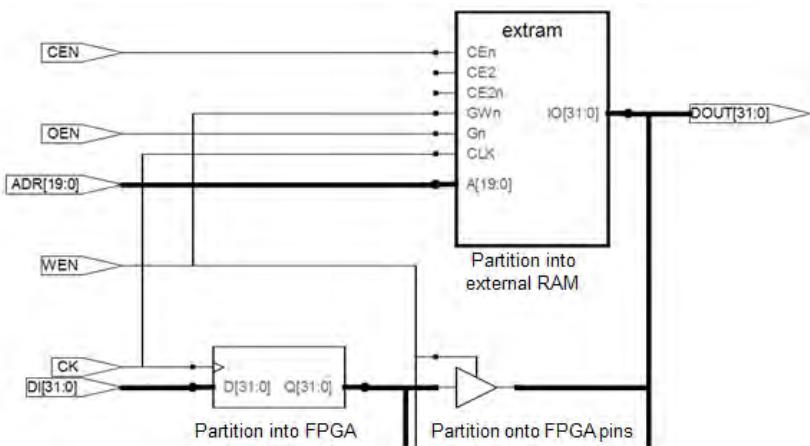


Note that the port names of the top level, shown as page connectors in the schematic, are the same as the ports on the wrapper. This is not strictly necessary but careful choice of port names will make it easier for others to understand the intent and also some tools will be able to make additional associations by name. For example, during partitioning, the Certify® tool can associate the port names of an instantiated black box within the wrapper, with the pin names of the external memory as described on the board description (see chapter 8)

However, a wrapper can be more sophisticated and can be used to manipulate the SoC top-level ports into something that connects with rather different FPGA or external resources. For example, a wrapper might be written to merge input and output buses on an SoC RAM instantiation, into a common tri-state bus for connection to an external SRAM, as shown in Figure 90.

Here a 1Mx32 SoC RAM cell is being modeled with a small external memory device, using the RTL shown in Figure 91. Again, the top-level ports correspond with the RAM instantiation in the SoC RTL, the lower pin on “extram” correspond with the pin names on the RAM device as they appear in the board description.

Figure 90: Wrapper merging SoC RAM data ports onto bidir port on external RAM



In this case both the RAM in the SoC and in the external device are named explicitly. We shall see later how we can make generic wrappers which allow parameterization and allow wider reuse for our wrappers.

Figure 91: VHDL code for wrapper shown in Figure 90 above

```
-- entity matches ram cell instantiation in ASIC design
entity UMC1048576x32S is -- 1Mx32 RAM
port (
    ADR : in std_logic_vector(19 downto 0);
    DI : in std_logic_vector(31 downto 0);
    DOUT : out std_logic_vector(31 downto 0);
    CK : in std_logic;
    WEN : in std_logic;-- active low
    CEN : in std_logic;-- active low
    OEN : in std_logic -- active low
    );
end UMC1048576x32S;
architecture wrap of UMC1048576x32S is
component extram is
port (
    A : in std_logic_vector(19 downto 0) ;
    IO : inout std_logic_vector(31 downto 0) ;
    CEn : in std_logic ;
    CE2 : in std_logic ;
    CE2n: in std_logic ;
    GWN : in std_logic ;
    Gn : in std_logic ;
    CLK : in std_logic ) ;
end component;

signal extrambus : std_logic_vector (31 downto 0);
signal regdi   : std_logic_vector (31 downto 0);
begin
process (CK)
begin
if rising_edge (CK) then
    regdi <= DI;
end if;
end process;
-- wrapper logic to combine/split input and output data onto bidir on external ram
extrambus <= regdi when WEN ='1' else (others=>'Z');
DOUT <= extrambus;

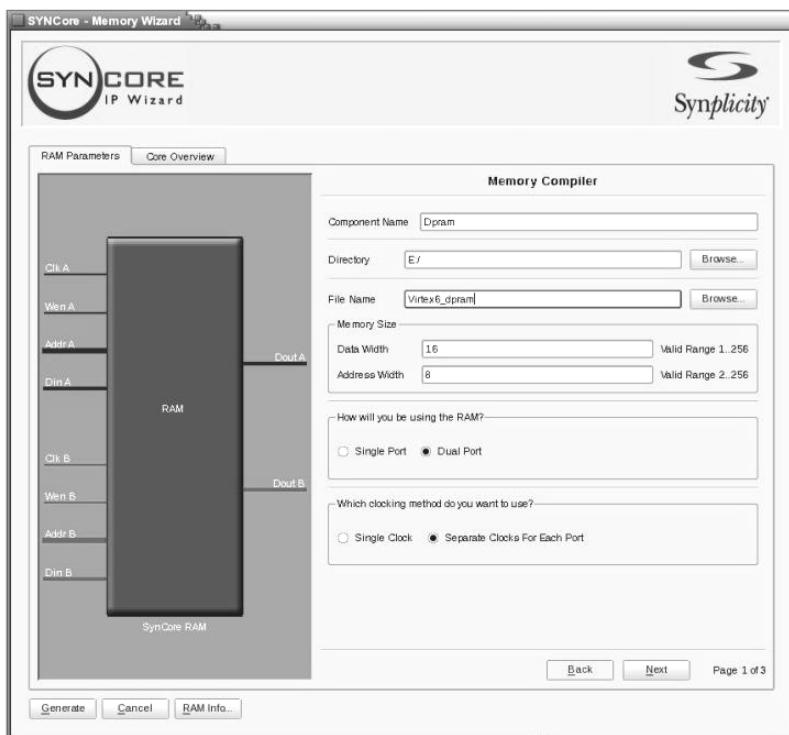
-- ram declaration matches ram chip on prototype board
UPD44322321: extram -- instance of external 1Meg x 32 Sync SRAM
port map (
    A => ADR,
    IO=> extrambus,
    CEn=> CEN,
    CE2=> '1',
    CE2n=> '0',
    GWN=> WEN,
    Gn=> OEN,
    CLK=> CK
    );
end wrap;
```

7.7.2.2. Tools for generating replacement memories

There are a number of tools which help to generate RAM and other memories for use in FPGA and we can use these for creating part of the contents for our wrapper. These tools are extensively used by FPGA designers for everyday production designs but can be equally useful for those using FPGAs only for prototyping. We will mention in particular two tools; CORE Generator tool from Xilinx and SYNCORE from Synopsys.

CORE Generator tool creates memory models for implementation only in Xilinx® FPGA, the flow is typically to use the black box instantiation of the memory as created by CORE Generator tool and then the implementation is added-in automatically during place and route. The implementation of the memory (i.e., to fill the black box) is in a Xilinx-specific object format, called ngc, and might even be encrypted. The contents may be used by the synthesis tool if they can understand the ngc format. The FPGA elements can then be inspected for timing or physical information, which are both useful during FPGA synthesis.

Figure 92: Synopsys SYNCORE memory compiler



As an alternative to CORE Generator tool, Synplify Pro from Synopsys includes a sub-tool called the SYNCORE IP Wizard. SYNCORE generates portable parameterized RTL for IP elements including RAMs in different configurations such as single-port RAM, dual-port RAM and byte-enabled RAMs. Figure 92 shows a screenshot of SYNCORE showing a dual-port RAM being created to target a Virtex-6 FPGA. In this case, the output is human readable Verilog RTL and so fully useable during all stages of FPGA synthesis and place and route.

Tools such as SYNCORE and CORE Generator tool allow us to quickly generate the necessary internal FPGA RAMs and other memories for modeling the SoC instantiated memories, via the use of suitable wrappers.

7.7.2.3. Wrappers to infer equivalent RAMs

So far we have used wrappers to instantiate equivalent memories in place of the SoC instantiated memory. The memories generated by SYNCORE, however, are actually in RTL from which FPGA synthesis can infer the required FPGA memory. This approach can be expanded to allow the creation of a small library of RTL descriptions which can be parameterized by the wrapper to create a large variety of different memories, for example, corresponding with the different types that we listed earlier in Table 16 (see page 195).

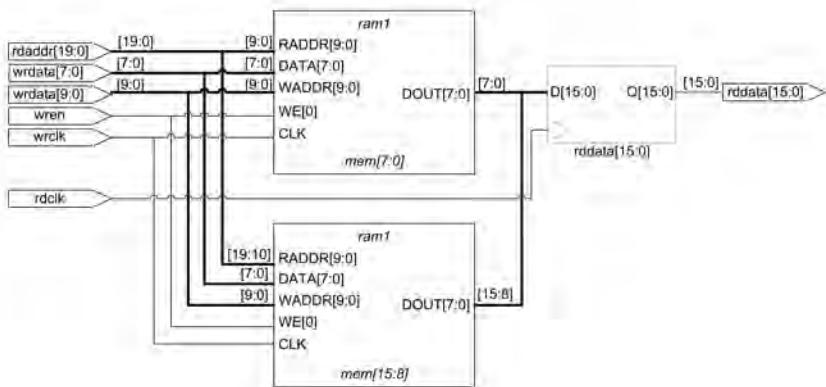
An RTL example for a parameterized RAM is shown in Figure 93 on the next page.

Figure 93: Example of parameterized generic RAM

```
module gen_ram #  
(  
parameter  
D_WIDTH    = 8,  
A_WIDTH    = 10,  
NUM_RDPRTS = 2 )  
(  
input wrclk,  
input wren,  
input [A_WIDTH-1:0] wraddr,  
input [D_WIDTH-1:0] wrdata,  
input rdclk,  
input [(NUM_RDPRTS*A_WIDTH) -1:0] rdaddr,  
output reg [(NUM_RDPRTS*D_WIDTH)-1:0] rddata );  
  
reg [D_WIDTH-1:0] mem [(1<<A_WIDTH)-1:0];  
integer i;  
  
always @ (posedge wrclk)  
begin  
  if(wren)  
    mem[wraddr] <= wrdata;  
end  
  
always @ (posedge rdclk)  
begin  
  for(i=0;i<NUM_RDPRTS;i=i+1)  
    rddata[i*D_WIDTH +: D_WIDTH] <= mem[rdaddr[i*A_WIDTH +:  
A_WIDTH]];  
end  
endmodule
```

This example is a single-write and multiple-read RAM but the number of read ports can be changed by the NUM_RDPRTS parameter. Notice that the defaults in this example are used to set the number of read ports to two, but this would be overridden by a new parameter passed into the RTL from the hierarchy layer above. Synplify Pro would synthesize the above RTL into the RAM structure shown in Figure 94.

Figure 94: RAM structure inferred by RTL in Figure 93



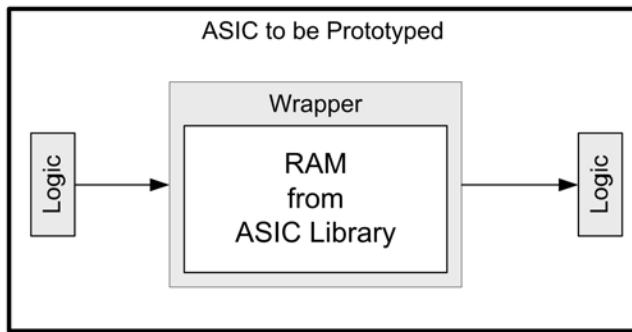
When mapped into BlockRAM in a Virtex-6 device, the rddata register bank would also be packed into the BlockRAM.

7.7.3. Advanced self-checking wrappers

We have so far considered two different kinds of wrappers. We have seen that some SoC designs do not use wrappers and instead instantiate the SoC memory directly into the surrounding RTL. In those cases, we need to use the SoC memory instantiation itself to define the top of the wrapper and place the FPGA or external equivalent in that.

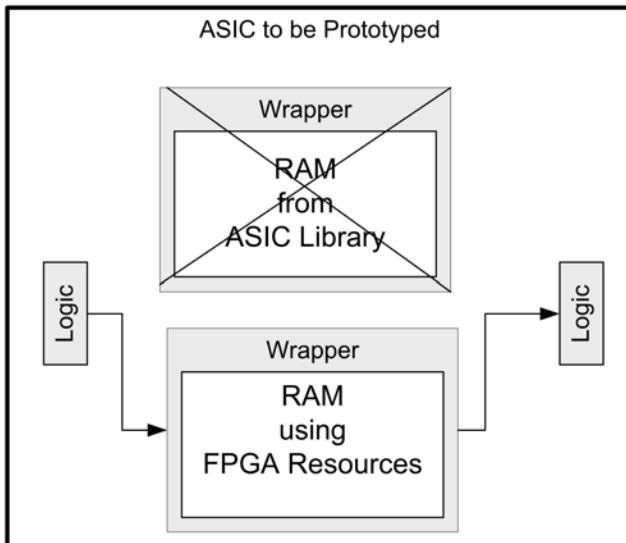
The second (and best) way to use memory in an SoC design is to put a wrapper around each instantiation, as shown in Figure 96. This requires that some foresight has been given to the needs of the prototypers and falls under the heading of Design-for-Prototyping, as we shall see in chapter 9.

Figure 96: Preferred wrapper in SoC design using Design-for-Prototyping



In normal prototype usage, we would replace the wrapper contents that instantiate the SoC memory with wrapper contents that instantiate or infer an FPGA equivalent, or an external chip, as shown in Figure 95.

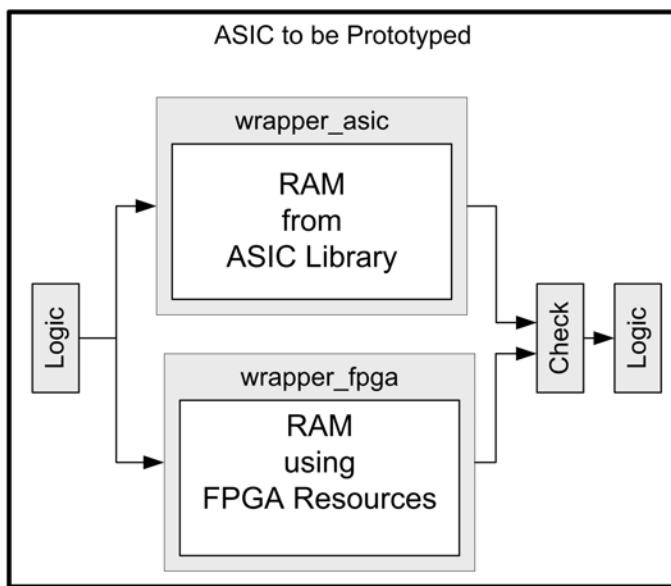
Figure 95: Switching between wrappers using `define fpga macro



Let's now consider a special case where we have a wrapper which instantiates both the SoC memory and the FPGA memory at the same time.

An overview of this arrangement is shown in Figure 97 where we can see that both wrappers are present throughout the verification process and we only choose one during synthesis using the branching macro again. During verification runs, both the FPGA memory model and the SoC memory model are evaluated and assertions are used to compare the output of each, which should be functionally identical. Only the SoC memory's result is passed to the rest of the logic. Within reasonable limits this should not drastically increase the simulation runtime but we do get the benefit that the FPGA memory is thoroughly tested in all SoC verification runs before being used in the prototype.

Figure 97: Self-checking RAM model



Using this approach, any discrepancies involved with assumptions on the RAM models between the SoC and FPGA versions can be found early in the design cycle, in fact even before synthesizing the design.

This methodology will require flow changes in the setup, and would definitely require that the SoC team embrace Design-for-Prototyping methods. Even if this requires some additional effort, we gain the advantage that memory modeling defects are found early in the design cycle.

We could also envisage a generic memory library in which for each memory used in SoC designs company-wide, we have a single file which encapsulates the RAM

from ASIC Library, the equivalent RAM using FPGA resources and the equivalency check. If we maintain such a generic memory library then we would not need to make any RTL changes for most memories when it comes to prototyping.

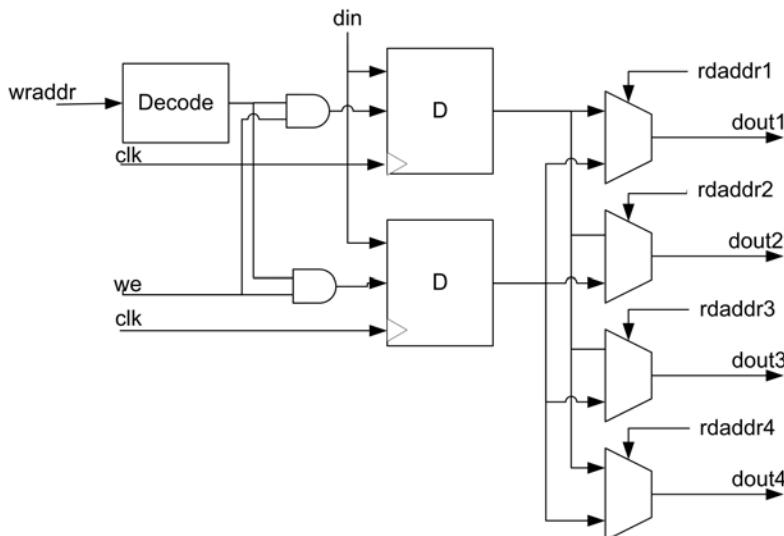
7.8. Implementing more complex RAMs

What if the RAMs instantiated in the SoC design are too complex to be mapped to dedicated BlockRAMs? Do not despair. With some ingenuity we can usually find a way to use the FPGA memories and other resources to mimic the behavior that we require. We can do this in extra RTL that resides inside the wrappers and therefore does not need a change to the SoC RTL. We cannot possibly explore all the possibilities here so we shall use two examples.

7.8.1. Example: implementing multiport RAMs

In the first case, let's consider a register files used in an SoC design, configured with one write port and four read ports. To implement this in an FPGA, the synthesis tools would map this register file into normal FFs in the FPGA logic

Figure 98: Four BlockRAMs used to implement a quad-port register file

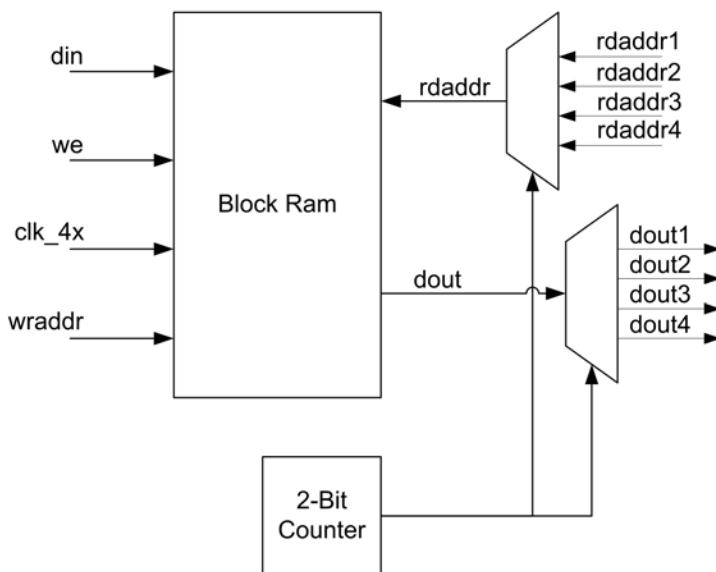


fabric. It would also use some LUT-based logic to perform decoding logic on the write port and for four multiplexers for the read ports. This may work very well for small register files but if such a structure were large enough, this could consume considerable amounts of logic.

Alternatively, we could envisage an implementation where four BlockRAMs are used in parallel, as shown in the schematic in Figure 98.

Whether or not this is really a better use of resources depends upon which resource we have most to spare, logic or BlockRAMs. However, there is an approach we can take which uses only a single BlockRAM as shown in Figure 99.

Figure 99: Single BlockRAM used to implement a quad-port register file



The BlockRAM and input and output multiplexers are clocked four times faster than the system clock. We then cycle a multiplexer at the higher clock rate to drive the four read ports using a four-to-one mux.

There would be a little extra logic overhead for muxing the read address and for generating the select signal for the output mux, but the overall resource usage would be lower than for either of the above implementations.

7.8.2. Example: bit-enabled RAMs

FPGA block memory resources are typically structured for byte-enabled writes so writing to a single bit would seem to be out of the question. However, there are a few techniques to build bit-enabled memories. One feature is to infer single-port and dual-port memories using the synthesis tool to map the bit-enables into the BlockRAM byte enables.

Figure 100, shows the parameterized model for a single-port memory with a

Figure 100: Excerpt from parameterized module to infer single-port RAM

```
module mem_sp_wren
#(parameter MEM_SIZE  = 512,
  parameter DATA_WIDTH = 15,
  parameter WR_SIZE   = 1,
  parameter SYNC_OUT   = 1'b1,
  parameter PIPELINED  = 1'b0,
  parameter RAM_STYLE  = "")
  (input          a_clk,
  input [DATA_WIDTH/WR_SIZE-1:0] a_wr_en,
  input          a_rd_en,
  input          a_reg_en,
  input [clogb2(MEM_SIZE)-1:0]  a_addr,
  input [DATA_WIDTH-1:0]         a_din,
  output [DATA_WIDTH-1:0]        a_dout
);

reg [DATA_WIDTH-1:0] mem[0:MEM_SIZE-1];
reg [clogb2(MEM_SIZE)-1:0] a_addr_r;
reg [DATA_WIDTH-1:0]      a1_dout;
wire [DATA_WIDTH-1:0]     a2_dout;
reg [DATA_WIDTH-1:0]      a3_dout;
```

parameter to set the write size. We can see that parameters can be passed into the model for setting all the usual dimensions of the RAM, with their default values in case of omission.

Figure 101: Wrapper for single-port RAM inference of SoC instantiation

```
module M256X10H1M8S10_wrapper (Q ,ADR ,D ,WEM ,WE ,ME ,CLK);
output [9:0] Q;
input [7:0] ADR;
input [9:0] D;
input [9:0] WEM;
input WE;
input ME;
input CLK
`ifdef FPGA
    mem_sp_wren
#(.MEM_SIZE(256),.DATA_WIDTH(10),.SYNC_OUT(1).PIPELINED(0))
    umem (
        .a_clk(CLK),
        .a_addr(ADR),
        .a_wr_en({10{WE}} & WEM),
        .a_rd_en(ME),
        .a_reg_en(1'b0),
        .a_din(D),
        .a_dout(Q)
    );
`else
M256X10H1M8S10 u_mem (.Q31(Q[31]), .Q30(Q[30]), .Q29(Q[29]),.....);
`endif
endmodule
```

Figure 101 shows a wrapper for a bit-enabled single-port memory which either instantiates the model above with relevant parameters or instantiates the SoC RAM cell depending upon the value of the FPGA macro. The parameters for the model can often be extracted from the original SoC memory name, which usually follows some logical pattern. In this case, 256X10 is the memory depth and data width, H1M8S10 defines a single-port bit-enabled memory.

We can see from the port map that there is a parameter called RAM_STYLE for specifying the FPGA memory type to which the RAM will be mapped, taking the possible values: BlockRAM, distributed RAM or registers. If left unspecified, as in this example, the synthesis tool will pick the most efficient style at the time that the RAM is inferred. Typically, if the memory size is less than 256 it will infer distributed memories.

Figure 102 shows the RTL to handle the configurable write enables which could be of any width write. We can work through the RTL to see how the write enabling on less than the full BlockRAM width is performed.

When we come to bit-enabled dual-port and multi-port memories, these cannot be directly inferred into FPGA memory structures but once again, with some ingenuity when creating wrapper contents, we can find techniques for creating equivalent functionality.

Our solution is a variation on the same technique used for standard multi-port memories that we described in section 7.6.1. Using a double-speed clock, a write cycle now consists of two fast clock cycles. On the first cycle, data is read from port

Figure 102: Excerpt of port logic within the model showing variable width write enable

```
integer i;
// A_PORT
always @(posedge a_clk) begin
    for (i=0; i<=DATA_WIDTH/WR_SIZE-1; i=i+1) begin :inst
        if (a_wr_en[i] == 1'b1) begin
            mem[a_addr][(i*WR_SIZE)+:WR_SIZE] <=
                a_din[(i*WR_SIZE)+:WR_SIZE];
        end
    end
    a_addr_r <= a_addr;           //register read add
    if (a_rd_en == 1'b1 && SYNC_OUT) begin // if sync out
        a1_dout <= mem[a_addr];          // read is on input
    end
    if (a_reg_en == 1'b1 && PIPELINED) begin
        a3_dout <= a2_dout;
    end
end

assign a2_dout = SYNC_OUT ? a1_dout : mem[a_addr_r];
assign a_dout = PIPELINED ? a3_dout : a2_dout;
```

B using the same address to which we wish to write. This read-back data is presented to a gating structure which will mix it with the new data to be written. Then, on the second cycle, the write data to put into the RAM is chosen dependent on the bit-write enable mask. There is another variation on this idea shown in Figure 103 but here a 1W 2R port memory is used to construct a single-port bit-enabled memory, but instead of a high-speed clock, it uses both edges of the clock, falling for reads, rising for writes.

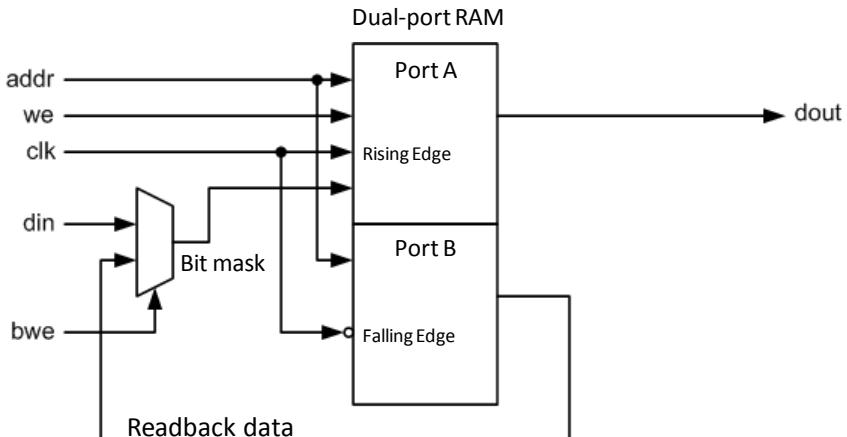
There are very many RAM topologies in use in modern SoC devices and the FPGA devices and tools infer many, but by no means all of them automatically. With our examples, we hope that we have illustrated that by good engineering we can always

find a way to model the SoC RAM in the FPGA fabric, or by using external components on the board.

7.8.3. NOTE: using BlockRAM as ROMs

One feature of RAMs supported in the FPGA architecture is that they can be configured as ROMs because the RAM can be pre-loaded with desired values. This can be done at configuration time so that the FPGA comes out of reset with the RAM contents already defined. So for all intents, we have a ROM. Since the RAM is a synchronous element, any ROM created from it will have the same restrictions.

Figure 103: Bit-enabled dual-port memory using both edges of write clock



So any function that requires a look-up list of values needs to also have its outputs registered. If these restrictions are met, then the function can be mapped into a BlockRAM, saving LUT resources in the logic fabric of the FPGA. In many cases the synthesis tool can infer the use of ROM in this way from a suitable RTL description.

7.9. Design implementation: synthesis

Once the design is made FPGA-ready, on the assumption that it fits into a single FPGA then we can move on to FPGA implementation, which as we saw in chapter 3, is comprised of synthesis and place and route (we shall deal in the next chapter about partitioning into multiple FPGAs, should that be necessary).

To recap, FPGA synthesis primarily compiles the RTL, checks for synthesis errors, accepts implementation constraints and generates FPGA netlists that are forwarded to the place & route tools along with further constraints.

To prepare our FPGA-ready design for synthesis, we must also enter the implementation constraints. These can be entered either in the design RTL itself or in the synthesis constraints files that can either be created directly by the user or by using a GUI provided by some tools.

The following are the most common implementation constraints:

- **Device properties:** these basic constraints direct the synthesis and subsequently the place & route tools to the FPGA family, specific device, speed grade and physical package.
- **Pin locations and properties:** this constraint maps each logical IO pin to a physical FPGA pin. The selection of pins is usually related to the prototyping platform architecture. If pin locations are not locked at this time then the pin assignment will change in subsequent synthesis runs and the design will not work in the FPGA. In addition to the physical pin location, we can specify IO properties such as signaling levels, slew rate, drive strengths etc.
- **Timing constraints:** this type of constraint is one of the most critical to the success of the prototyping effort. These constraints drive the synthesis and are passed on to the place & route tools to drive each towards the desired timing, balancing resource utilization and implementation effort. Timing constraints are usually entered in a synthesis constraints file, either manually or by using a GUI. Effective timing constraints will guide the implementation tools to put the effort where it is most needed. The most common timing constraints are: clock period, from/to delays, and multi-cycle paths.
- **Other constraints:** there are many other constraints and attributes that guide the synthesis and place & route tools, such as synthesis styles, state machine implementation styles etc. Good knowledge of the available attributes and how to apply them will help us to best leverage the tool and device capabilities to meet our prototyping goals.

7.9.1. Note: using existing constraints for the SoC design

In some cases the timing constraints for the original SoC design might be reusable to some degree in the FPGA-based prototyping flow. SoC synthesis tools, such as Design Compiler® (DC), will usually be configured in their project scripts to perform a bottom-up synthesis and so constraints are applied at each block level. However, there may often be top-level constraints which give at least the IO

constraints for the device. There may even be a top-down flow employed in smaller designs, in which case the top-level timing constraints may be all that are available. In either case, the constraints will usually be written in a format called standard design constraint or SDC for short.

Advanced FPGA synthesis tools can make use of common timing constraints in the SDC format, thus allowing direct reuse of the same SoC constraints for the FPGAs in the prototype.

As an example, the following common design constraints are supported in Synopsys FPGA synthesis tools:

```
create_clock  
create_generated_clock  
set_clock_groups  
set_input_delay  
set_output_delay  
set_false_path  
set_multicycle_path  
set_max_delay
```

These are the common set of timing constraints used in SoC designs and are fairly self explanatory from their names. Other unsupported constraints should either be manually translated (if a corresponding constraint is present in the FPGA synthesis tools) or ignored, depending upon importance. The tool documentation will give guidelines of the level of SDC support provided.

The SoC SDC will usually include much more than top-level timing constraints, for example “report” generation commands. Typically these are not directly supported or have a different syntax in the FPGA tools, so equivalent reports must be generated by using the timing analyzer capability within the FPGA synthesis and place & route tools manually.

To use the SDC timing constraints for FPGA synthesis in Synopsys FPGA synthesis tools, we need to take care of the naming rules employed. Names may change or be ambiguous between different tools, so some mismatch might occur between common naming conventions. All naming rules are configurable in Design Compiler and there are a wide variety of styles adopted by different SoC teams. For example, in the notation of hierarchy separators, naming of bus signals, multidimensional arrays, structures, records and generated signals and identifiers. If the naming conventions in the Design Compiler SDC do not match the default naming conventions followed in the FPGA synthesis tools, then we can explicitly add the appropriate command specifying the naming conventions in the beginning of the SDC file.

7.9.2. Tuning constraints

Tools will often have a constraint checker to allow a quick-pass analysis of coverage and legality of the constraints before running synthesis or place & route. The results are presented in a report which provides information on how the constraints will be interpreted by the tool, without having to wait for the tool to complete a full run. Based on the report, we can quickly edit the constraint file rather than wade through messages in a synthesis log file, which might run into many thousands of lines.

When synthesis is complete, the tools will provide reports giving details of utilization and estimated timing. These reports can give an early warning of design issues such as inter-FPGA critical paths, unexpected utilization levels and missing components. Although all timing is based only on estimates, synthesis reports should be carefully examined before proceeding to the place & route process as the design or constraints that may need to be modified. To ease this process, the reports can be examined automatically, either using the tool's built-in report features or using scripts which search and manipulate the report files directly.

7.10. Prototyping power-saving features

In most cases, the power mitigation of an SoC is implemented by the use of scripts and control files outside of the RTL, in a language such as universal power format (UPF). The result of running UPF controls with SoC synthesis and back-end are that the RTL is modified and/or supplemented with extra circuitry. Generally, the prototyping team receives the RTL before this power mitigation work has taken place and it is still a pure description of the SoC function for prototyping purposes.

However, with the growth in sophistication of SoC power mitigation techniques, prototypers are sometimes asked if there is a way to test the RTL after the UPF and similar controls have inserted power-down modes, data retention and clock control.

Power modeling in FPGA is an interesting problem and not something that is easily implemented. Some functions obviously do not make sense in the FPGA, for example, the modeling of switching off the power to certain parts of the chip, but there is merit in prototyping some of the other properties.

It is possible to use the clock-gating features of synthesis tools to model some of the behavior, and careful use of PLLs can allow the frequency scaling features to be implemented, but adjustment of the supply voltage would not give any benefit, nor would the power monitoring of the core supply to the FPGA to determine power consumption as this would not be representative, and even worse give misleading results

However, it may be beneficial to use our prototype to model the power-control system itself and for this we would need to create some extra RTL, which would fool the power controls and the software into believing that the silicon was responding as expected. We can indeed dynamically adjust the frequency via internal or external PLLs. We could also mimic the expected responses of a voltage change or the effects of turning-off sections of the design. This might be useful for validating the software that controls such things but the complexity might far out way the benefit.

The model in its simplest mode would mimic the registers and interfaces seen in the real world and implement the time delays we would expect for the real system. However power gating sections of the design is more difficult as we would need to generate the impression that a section is powered down when the FPGA silicon cannot actually be powered down. For example, we would need to implement clock gating or other features to ensure that a section does not respond when it is meant to be switched off. It would also be impossible to represent the random values that might appear in SoC registers that are powered down because the FPGA registers would not actually have been powered down and will still hold valid values, perhaps masking an effect that would be very important to witness in the real sislicon. In these examples we see that perhaps, modeling SoC power-down feature s in FPGA might be more misleading than helpful.

To implement these features would either require RTL modification to the real SoC design (which is not ideal and should only be considered if really necessary) or the use of the ability of the Xilinx® FPGA device to be reconfigured (partial bitstreams with different INIT values may be loaded while the devices are operating, to determine if the changes in the values affect the design). It is not the aim of this book to explain partial reconfiguration, but further information can be found in the references.

Recommendation: prototyping is intended for creating a functional model of an SoC for software validation and system-level integration of software and hardware, pre-silicon. It performs this task best upon RTL which is delivered before it is manipulated for power saving or test purposes.

7.11. Design implementation: place & route

Place & route tools take the FPGA-level netlist, implementation constraints, and tool-specific directives generated by the synthesis tool, and perform mapping, and place & route to produce a bitfile to download into the FPGA on the board. If the design was properly constrained in the synthesis phase all implementation constraints will flow from the synthesis tool to the place and route tools. The user can provide a number of tool directives such as operational modes, effort levels, and reporting details to optimize its operation and reporting.

During the place & route process, the tools provide various intermediate status reports. Since for large designs the place & route process can take hours to complete, it is recommended that we review these intermediate reports as they become available and determine if the process is progressing as expected or if an intervention is needed. The following is a list of the critical reports typically generated through the place & route process:

- **Mapping and utilization reports:** these reports provide an account of the resources found in the design. These reports may indicate problems in the design if logic was unexpectedly removed etc. This can happen due to improper inclusion of cores as black boxes, or a cascading effect due to the removal of other blocks. To rectify such situations we should go back to the design and verify that the removed blocks are properly connected to the design after synthesis, that all cores are properly instantiated and all cores' netlist files are available to the place & route tools.
- **Preliminary timing report:** usually before the full place & route process takes place, the tools estimate if the timing constraints are possible for the design. If for example, the number of combinatorial logic levels between two FFs is such that the time it will take to propagate through them is greater than the requested clock period, even with minimal routing delays, the tool may stop processing the design. In such case we need to understand the critical timing violations and determine if paths can be multi-cycle paths or make design changes to shorten them.
- **Final timing report:** this is the ultimate timing report for the design after the place & route process is complete. This report should be used to determine if the design meets the desired timing. If the timing results do not meet the requirements, the prototyping engineer needs to understand which paths are too long and rectify the issues either through tools' performance parameters, tighter constraints, or design changes. It's important to note that the timing reports are for each FPGA in isolation, and external timing analysis needs to be done, such as FPGA to FPGA and FPGA to other hardware. As mentioned in the "Design modifications due to prototyping system" section in this chapter, inserting FFs at all FPGA IOs will significantly optimize and simplify board-level timing calculations.

There are other manuals regarding the place and route of FPGA designs, including those in the references. In general, the default operation of place and route may suffice for some projects but as with all tool flows, exceptions will occur that require us to "roll up our sleeves" and become expert in some aspects of some tools. As discussed in chapter 4, FPGA-based prototyping is most successful when performed by FPGA experts.

7.12. Revision control during prototyping

As we have seen, although we try to minimize the impact of FPGA-based prototyping on the SoC source, we will probably need to make changes to the design files. As with all engineering tasks it is crucial to track and document these changes. In addition there may be changes to the embedded software running in our prototype or in simulation testbenches and of course, the prototyping tool set-up, scripts and so forth will need to be recorded in order to be repeatable. For all these reasons, the use of a revision-control system during prototyping will greatly assist us in recreating the prototype across platforms, sites and future derivative projects.

Undoubtedly, our labs already use revision control for hardware and software projects, and tools such as Perforce are widely used at Synopsys and Xilinx. When delivering RTL for use by the prototyping team, this is no less important to track than any other branching of the code. In parallel, the embedded software branches to run on the prototype may be very similar to that which will eventually run on the SoC (we certainly hope so) but there will be small changes, for example, a time constant in a header file to account for a slower clock. These small software changes must also be controlled and only the appropriate changes used for a particular prototype build.

This becomes exponentially more difficult to control when multiple engineers are working on the prototype and time is short. If we make changes to our branch of the SoC source code to enable prototyping or debug, then these will probably not be wanted back in the mainline SoC code repository. However, a change in an RTL file to fix a bug discovered during prototyping must obviously be fed back into the mainline code. Only good revision control will enable us to keep track and discriminate between these two cases.

We must resist the temptation to make quick and temporary changes during prototyping even though FPGAs offer great freedom to make exactly these kinds of quick changes. If we work with the mindset that anything we do can impact the final silicon, even though we are not working on the mainline of the code (either RTL or software), then we can avoid much unnecessary, inefficient and perhaps ultimately costly confusion.

7.13. Summary

In this chapter we have covered the bulk of the tasks we undertake in order to overcome the third law of prototyping. We have used a number of techniques to remove or at least neutralize elements of the SoC design which would not have worked as they are in FPGA. We hope that we have not discouraged readers from starting because even the most pathologically FPGA-hostile design can eventually be mapping onto an FPGA board.

Successful SoC prototyping demands a good understanding of FPGA technology and implementation tools, but most of all, we need to recognize which parts of the design will best respond to one or other of the above techniques.

The authors gratefully acknowledge significant contributions to this chapter from:

Steve Ravet of ARM, Austin

Joel Sandgathe of Microvision, Seattle

Pete Calabrese of Synopsys, Boston

Ramanan Sanjeevi Krishnan of Synopsys, Bangalore

Nithin Kumar Guggilla of Synopsys, Bangalore

After following the guidelines in chapter 7, our design will be ready for FPGA, or should we say, ready for one FPGA. What if our design does not fit into a single FPGA? This chapter explains how to partition the FPGA-targeted part of our SoC design between multiple FPGAs. Partitioning can be done either automatically or by using interactive, manual methods and we shall consider both during this chapter. We shall also explain the companion task of reconnecting the signals between the FPGAs on the board to match the functionality of the original non-partitioned design.

8.1. Do we always need to partition across FPGAs?

Many FPGA-based prototypes will use a single FPGA device, either because the design is relatively small or because as prototypers, we have purposely limited the scope of our project to fit in a single FPGA. As explained in chapter 3, FPGA capacity keeps increasing in line with Moore's law, so one might assume that eventually all prototypes will fit into a single device. However, SoC designs are also getting larger and typically already include multiple CPUs and other large-scale application processors such as video graphics and DSPs, so they will still overflow even the largest FPGA.

As we shall see in chapter 9 on Design-for-Prototyping, an SoC design can be created to pre-empt the partitioning stage of an FPGA-based prototyping project. The RTL is already pre-conditioned for multiple devices and we can consider the SoC design to consist of a number of separate FPGA projects. By planning in advance we can ensure that each significant function of the SoC design is small enough to fit within a single very large FPGA, or otherwise easy to split into two at a boundary with minimal cross-connectivity.

However, some designs do not have such a natural granularity and do not obviously divide into FPGA-sized sections so for the foreseeable future, therefore, we should usually expect to partition the design into more than one FPGA and for some designs this may be a significant challenge. In addition, not only do we need to partition the design but we also need to reconnect the signals across the FPGA boundaries and ensure that the different FPGAs are synchronized in order to work the same as they will have in a single SoC. Let's look in turn at partitioning, reconnection and design synchronization.

8.1.1. Do we always need EDA partitioning tools?

If we have not designed our SoC expressly for multiple FPGA prototyping, it is unlikely that we will be successful without using EDA tools to either aid or completely automate the partitioning tasks. There are a number of EDA tools that aid in the partitioning effort and greatly simplify it. Generally, these tools take as their inputs the complete design and a system resource description, including the FPGAs, their interconnections, and other significant components in the prototyping system. Some are completely script driven based on a command file, while others are more interactive and graphical. Interactive tools display the design hierarchy and the available resources and allow us to drag-and-drop design elements of sub-trees “into” specific FPGAs. Advanced tools will dynamically show the impact of logic placement on utilization and connectivity.

The following list depicts the advantages of using EDA tools to perform partitioning:

- Global implementation constraints are possible and the tools transparently propagate these constraints to the place & route tools for each FPGA.
- The partitioning tools will optionally insert pin multiplexing and employ logic replication as needed.
- Some partitioning tools are tightly integrated with FPGA synthesis tools allowing resource and timing information to be generated by one and used by the other. This integration further simplifies the partitioning and synthesis processes.

8.2. General partitioning overview

The quality of the partitioning can significantly affect the resulting system size and performance, especially for large and complex SoC designs.

As we saw in chapter 3, there are a number of approaches to partitioning giving us the general choices of partitioning at the RT level before synthesis or at netlist-level after synthesis. In either case, the same general guides for success apply. Partitioning is performed in a number of stages and requires some advance planning. The prime goal of partitioning is to organize blocks of the SoC design into FPGAs in such a way as to balance FPGA utilization and minimize the interconnect resources. Therefore we need to know details of both the size of the different design blocks and also the interconnections between them. We shall consider that in a moment.

8.2.1. Recommended approach to partitioning

Let's now look at the recommended order in which we should partition the FPGA-ready part of the design. Table 17 summarizes the steps to be taken during an interactive approach. We will consider each in detail in the coming sections and then go on to consider automated partitioning.

Table 17: Major steps in interactive manual partitioning

Task	Main reasons
Describe target resources	Basic requirement for all EDA partitioning tools. (even a manual approach needs pin location list).
Estimate area	Avoids overuse of resources during partitioning.
Assign SoC top-level IO	Ensures connectivity to external resources. Directs the choice of FPGA IO voltage regions.
Assign highly connected blocks	Minimizes inter-FPGA connectivity for large buses. Groups related blocks together for higher speed.
Assign largest blocks	Gives early feedback on likely resource balancing. Leaves smaller blocks for filling in gaps.
Assign remaining blocks	Allows manageable use of replication and pruning. Allows manageable working at lower levels.
Replicate essential resources	Reduces required amount of RTL modification Sync clock and reset elements in each FPGA.
Multiplex excessive FPGA interconnect	Frees up FPGA IO for critical paths. May be only way to link all inter-FPGA connections.
Assign traces	Assigning inter-FPGA signals fixes pin locations. Every FPGA pin location must be correct.
Iterate to improve speed and fit	Initial partition can usually be improved upon. Target speed can often be raised as project matures.

8.2.2. Describing board resources to the partitioner

In the meantime, we will already have an understanding of the board resources onto which we will need to map the design. We know the size of the FPGAs and have a complete list of the interconnection between them on our board. The partitioning tool will need to have this information presented in the correct format and in the case of the Synopsys Certify® tool, this is called a board description file and is written in Verilog. An excerpt from a board description file is shown in Figure 104, in which we can see clock traces, other traces and an instantiation of a Xilinx® LX760 FPGA.

Figure 104: Excerpt from typical board description for partitioning purposes

```
 . . .
 . . .
//wire [11:1]      A_GCLK0;           // board traces
//wire [11:1]      B_GCLK0;           // board traces
//wire [11:1]      C_GCLK0;           // board traces
//wire [11:1]      D_GCLK0;           // board traces

// wire [32:1]      SMAP_A, SMAP_B, SMAP_C, SMAP_D;
// board traces to each device

wire [1:1]         A_RESET_n, B_RESET_n, C_RESET_n, D_RESET_n, RESET_n;
wire [1:1]         A_RESET_INT_n, B_RESET_INT_n, C_RESET_INT_n,
D_RESET_INT_n;

// Inter-FPGA traces
wire [44:1]        AB;
wire [29:1]        AC;
wire [62:1]        AD;
wire [62:1]        BC;
wire [29:1]        BD;
wire [65:1]        CD;
. . .
. . .
// Device A Virtex-6 LX760

XC6VLX760FF1760 uA (
.pin_AR8          ( A1_A[1] ),
.pin_AT7          ( A1_A[2] ),
.pin_AM11         ( A1_A[3] ),
.pin_AN10         ( A1_A[4] ),
. . .
```

It is not necessary to describe parts of the board that do not affect signal connectivity, so items such as power rails, pull-up resistors, decoupling capacitors

or FPGA configuration pins need not be included. It is most important that the connectivity information is absolutely correct and if possible, the Verilog board description should be obtained directly from the original board layout tool. A script can be used to remove unwanted board items and to ensure correlation between the traces on the board and the wires in the Verilog.

Some boards and systems will have methods for generating this board description automatically, reflecting any configurable features on the board or daughter cards connected. For example, some systems have deferred interconnect or switched interconnect, as discussed in chapter 6, and so the board description will need to reflect the actual configuration of the board in our project. We will come back to consider adjustable interconnect in a moment.

Considering the area and interconnect requirements for the design as a logical database, and the board description as a physical database, then our partitioning task becomes a matter of mapping one onto the other and then joining up the pieces. If we take a step-by-step approach to this, we should achieve maximum performance in the shortest time.

8.2.3. Estimate area of each sub-block

In chapter 4 we used FPGA synthesis very early in the project in order to ascertain how many FPGAs we need for our platform. For partitioning, we need to know the size of each block to be partitioned in terms of the FPGA resources. We can do this by analyzing the area report for the first-pass synthesis either manually or by using grep and a small formatting script to create a block-by-block report. However, partitioning tools usually make this process more productive, and provide a number of ways to estimate the block area for LUTs, FFs and specific resources (e.g., RAM, DSP blocks). Most importantly, the tools should also estimate the boundary IO for each block and its connectivity to all other blocks.

Rather than running the full synthesis and mapping to achieve area estimates, we can optionally perform a quick-pass run of the synthesis or a specific estimation tool. For example the runtime vs accuracy trade-offs for the Certify tools are shown in Table 18.

The Certify tool can make these estimates, but more importantly, will also display and use those estimates during the partitioning procedures, as we shall see in a moment.

Table 18: Options for resource estimation in the Certify tool

Effort Level	Estimation Mode	Description
Low	Model based	Uses netlist manipulator to write estimation (.est) file; provides fastest execution time.
Medium	Model based	Uses netlist manipulator to write estimation file; improved estimation of registers over low effort but longer execution time.
High (default)	Architecture based	Uses FPGA mapper to write estimation file; requires longest execution time.

The result of estimation will yield values which are general very accurate for IO count, but the area will probably be an over-estimate. However, a good by-product of this over-estimation is that it encourages us to leave more room in our FPGAs. The results of a quick-pass estimation may be displayed in many formats, including text files which can be manipulated and sorted using scripts to extract important information. However the most useful time and place to see the results is during the partitioning, in order to prevent us making poor assignment decisions which cause problems later in the flow.

8.2.4. Assign SoC top-level IO

Certain external resources or connectors on the board may need to connect with specific blocks of the design, for example a RAM daughter card will need to connect with the DDR drivers in the design. This will dictate that the RAM subsystem will need to be partitioned into that FPGA which is connected to the RAM daughter board. Other pins of this kind may be connected to external PHY chips, or to test points or logic analyzer header on the board that will monitor certain internal activity.

The location of these kinds of fixed resources should be assigned first, since they are forced upon us anyway. Depending on how flexible your platform may be, it might be possible to alter which FPGA pins connect to these external resources by rearranging the board topology, for example, by placing daughter cards in a different place on the motherboard.

Some teams find that it helps to configure the platform so that one FPGA drives as much of the external SoC IO as possible. This may seem to over-constrain one

FPGA but the freedom it gives to assignments in the remaining FPGAs is very helpful.

8.2.4.1. Note: take care over FPGA IO voltage regions

In a typical SoC there will be ports that run at different voltage levels in order to connect to external resources. Consequently in our prototype, we need to configure our FPGA pins so that they can interface at the same voltages. FPGA pins are very flexible and can be set to different voltage standards but they must be configured in banks or regions of the same voltage rather than by individual pin, as we learned in chapter 3

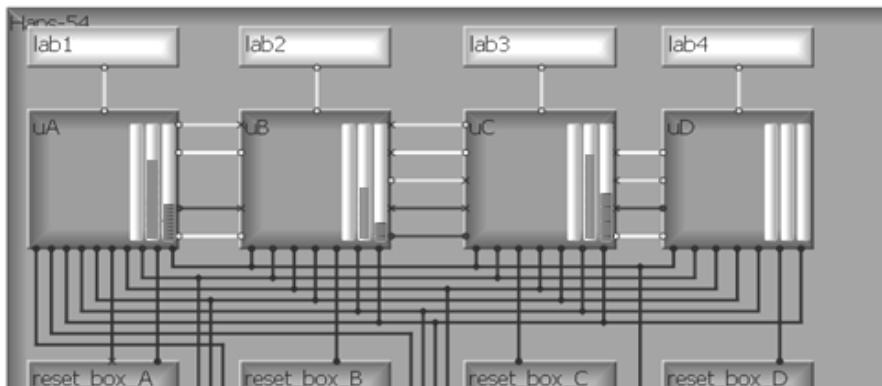
We therefore do not have complete freedom in our FPGA pin placement, so we should assign SoC ports that require specific voltage pins next, although it will still be possible later to move pins within the given voltage region if necessary. For example, in the Virtex®-6 family there are 40 pins in a voltage bank. It is very useful if the partitioning tool gives guidance and feedback on voltage regions and required voltages of the IO while we are partitioning and warns against incorrect IO placements or configuration,

8.2.5. Assign highly connected blocks

We can now start assigning the blocks of the SoC design into specific FPGAs. This first partitioning operation will steer all remaining decisions. As in most projects, it is important to make a good start. We begin with those blocks which share the most interconnect with other blocks, but how do identify those?

It is useful before and during partitioning to compare notes with the SoC back-end

Figure 105: Immediate indication of resource and IO usage in each FPGA



engineers who floorplan and layout the IC itself. They will probably be working through a trial implementation of the chip around the same time as the prototype is being created (see chapter 4). Both teams face the same partitioning challenges, especially if there are areas of congestion and high-connectivity in the RTL.

The RTL designers or back-end team may also provide connectivity information, but in many cases we have to ascertain this by inspecting the block-level interconnect information ourselves. This is where some tools can help by displaying or ordering this information for easy inspection, for example, by overlaying the block-level area and interconnect information onto other design views. For example, Figure 105, shows a board view from the Certify partitioning environment where we can see indicators of IO and Logic usage for each FPGA. Interconnect information is also crucial and Figure 106 shows the Certify tool's display of block-level interconnect data at a particular level of the design hierarchy.

Figure 106: Mutual interconnect between blocks displayed as a matrix

	Visible Ports	Hidden Nets	b2v_inst	b2v_inst1	b2v_inst14	b2v_inst2	b2v_inst3
Visible Ports	144		72	4	38	79	106
Hidden Nets			0	0	0	0	0
b2v_inst	72	0		0	35	36	36
b2v_inst1	4	0	0		0	82	0
b2v_inst14	38	0	35	0		0	0
b2v_inst2	79	0	36	82	0		11
b2v_inst3	106	0	36	0	0	11	

Here we can immediately see that blocks b2v_inst1 and b2v_inst2 share 82 mutual connections but that b2v_inst3 has the most connections (106) to the block's top-level IO.

The important task when partitioning multiple blocks with large numbers of mutually connected signals is to ensure that these blocks are placed in the same FPGA. If highly connected blocks are placed in different FPGAs then we will need a large number of FPGA IO pins to reconnect them. For example, when using 64 bit and larger buses, it is quite possible that two blocks assigned into different FPGAs can require hundreds of extra FPGA IO. So in our example above, we might try to assign b2v_inst3 first into one FPGA while b2v_inst1 and b2v_inst2 can be assigned together into a different FPGA because they are mutually connected, but share little connectivity with b2v_inst3.

If it is not possible to put highly-interconnected blocks together because they overflow the resources of one FPGA, then we will need to step down a level of hierarchy and look for blocks at the next level which are less connected and extract those to be assigned in a different FPGA. In this way we may still increase the number of required FPGA IO, but by less than would be the case if the higher-level block were assigned elsewhere. If there is no such partition at this lower level then

we might go even lower, but specifying partitioning at finer and finer logic granularity makes it more likely that the partition will be affected by design iterations as these finer grains are optimized differently or renamed. If we find ourselves having to go deep into a hierarchy to find a solution then it may be better to go back and restart at the top-level with a different coarse partitioning.

Recommendation: try different starting points and partially complete the assignments to “get a feel” for the fit into the FPGAs and interconnect. If a partitioning effort quickly becomes hard to balance then they are unlikely to complete satisfactorily.

In cases of IO overflow or resource overflow, it is useful to have immediate feedback that this is happening as we proceed through our partitioning tasks. As an example, when block assignments are made to FPGAs in Certify partitioner, we see immediate feedback on IO and resources usage in a number of ways, including the “thermometer” indicators on the FPGAs of the board view, as shown in Figure 105. Here we can see a graphical representation of the four FPGAs on a HAPS® board and in each FPGA, there are three fields showing the proportion of the internal logic resources and the IO pins used in this partition so far. As we assign blocks to each FPGA we will see these thermometers change up, and sometimes down. A glance at this display tells us that the IO and logic usage is well within limits.

8.2.6. Assign largest blocks

Using our estimation for the area of each block, we can assign the rest of the design blocks to FPGA resources, starting with the largest of the blocks. We start with the larger block because this naturally leaves the smaller blocks for later in the partitioning process. Then, with the FPGA resources perhaps becoming over-full (remember 50% to 70% utilization is a good target) we have more freedom in the placement of smaller blocks of finer granularity and lower number of inputs and outputs.

As we partition, we look to balance the resource usage of the FPGAs while keeping the utilization within tolerable limits i.e., less than 70% recommendation. This will help avoid long place & route runtimes and make it easier to reach required timing.

Recommendation: if there is a new block of RTL which is likely to be updated often or to have its instrumentation altered frequently, then keeping utilization low for its FPGA will speed up design iterations for that RTL and save us time in the lab.

As each block is assigned, we may find that the available IO for a given FPGA is exceeded. The remedy for this is to go back and find a different partition or replicate some blocks (see below) or consider the use of time-division multiplexing of signals onto the same IO pin (see also below). At all stages the feedback on current

utilization and IO usage will help us to make immediate decisions regarding all the above items.

8.2.6.1. Note: help assigning blocks

Having selected a candidate block for partitioning we might make trial assignments until we find the best solution, however, that is inefficient in a prototype with many FPGAs. We have seen how useful it is to have immediate feedback on our assignment decisions. In fact, it is even more useful to have the feedback before the assignment is made. This allows us to see in advance what would be the effect on IO and resources if a selected block were to be placed in such-and-such FPGAs. This kind of pre-warning is called impact analysis.

In the case of Certify, impact analysis can instantly make the trial assignment to all FPGAs on our behalf and then show us the impact in a graphical view, as shown in Figure 107.

Figure 107: View of impact analysis in Certify tool

Area/IOS			
Device	Calc	Area	IOs
mb.uA	<input checked="" type="checkbox"/>	693 : 21 / 478080 (672)	100 : 183 / 1200 (-83)
mb.uB	<input checked="" type="checkbox"/>	672 : 0 / 478080 (672)	150 : 13 / 1200 (137)
mb.uC	<input checked="" type="checkbox"/>	672 : 0 / 478080 (672)	139 : 0 / 1200 (139)
mb.uD	<input checked="" type="checkbox"/>	672 : 0 / 478080 (672)	139 : 0 / 1200 (139)

Calculate
Assign
Show insts...
Advanced

Here we can see that our selected block has an area of 672 logic elements, extracted from a previous resource estimate. If we choose to assign our block to mb.uB, we will increase that FPGA's area by 672 logic elements (out of a total of 478080) and we will increase the IO count by 137, bringing it to a total of 150. We can also see that if we assign our block to mb.uA, then the area will still increase by the same amount but the IO requirement will fall by 83 pins, presumably because our block connects to some logic already assigned to mb.uA. We can select mb.uA based on this quick analysis and then click assign.

As with all tools driven by an interactive user interface it is good to be able to use scripts and command line once we are familiar with the approach. In the case of Certify's impact analysis, a set of TCL commands is available to return the

requested calculation results on the specified instances for the indicated devices. Results can be displayed on the command line or written to a specified file for analysis.

This semi-automated approach to block assignment leads us to ask why a fully automated partitioning would not be able to perform the same analysis and then act upon the result. We shall look at automated partitioning in a later section.

8.2.7. Assign remaining blocks

After the major hierarchical blocks have been placed, we can simply fill in the gaps with smaller blocks using the same approach. The order is not so important with the smaller blocks and we can be guided by information such as connectivity and resource usage. Some tools also offer on-screen guidance such as “rats-nest” lines of various weights based on required connections which appear to pull the selected block towards the best FPGA to choose.

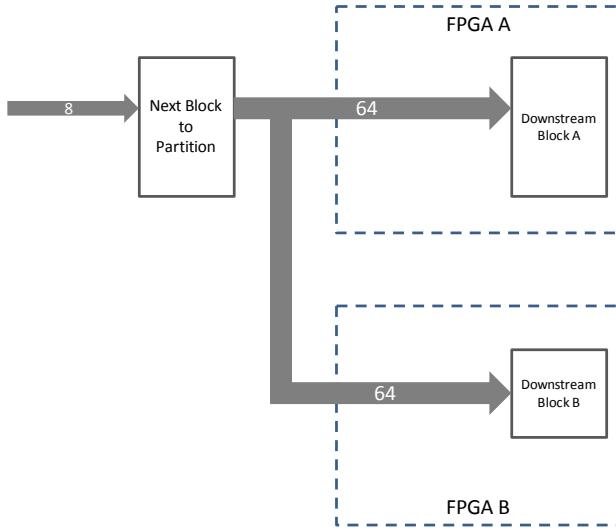
Having made their key assignments manually, some teams will switch to using automated partitioning at this stage. If this can operate in the same environment as the manual partitioner then that is more efficient. We simply get to a point where we are satisfied that we have controlled our crucial assignments and push a button for the rest to be completed. For example, Certify’s quick partitioning technology (QPT) can be invoked from within the partitioning environment at any time.

8.2.8. Replicate blocks to save IO

While partitioning a system as large as a complex SoC we are likely at some time to reach a point where a block needs to be in two places at once.

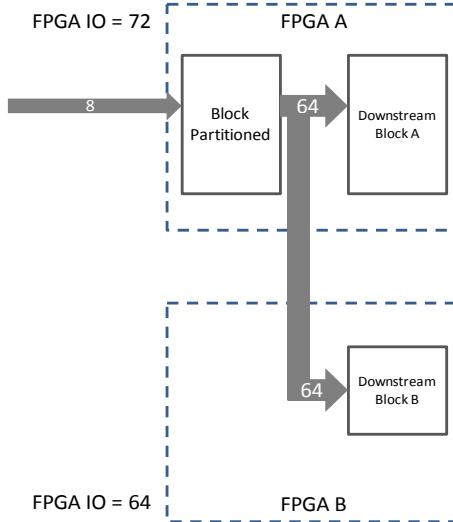
For example, the next block we wish to partition needs to drive a wide bus to two blocks which were previously partitioned into different FPGAs, as shown in Figure 108.

Figure 108: A partitioning task; where to place the next block?



Which FPGA is the right place to put our next block? Either choice will cause thereto be a large number of IO which need to leave the chosen FPGA to connect the bus across to the un-chosen one, as shown in Figure 109.

The answer is to put the block into both FPGAs as shown in Figure 110. In this way each downstream block has its own copy of the new block inside its FPGA and so Figure 109: Partitioning into either FPGA requires 136 extra IO pins

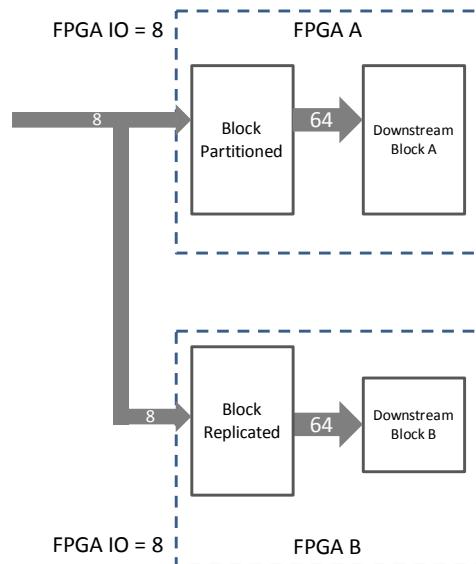


no IO is required except for that which feeds the new block, IO requirement drops from 136 to 16.

Our partitioning tool should allow us to make multiple copies of the same block for partitioning into different FPGAs. If not, then it might mean a lot of rewriting of RTL, which may not be practical, even with the use of XMRs and other short cuts.

Partitioning tools will have different ways to achieve this replication but in the case of Certify it is simply a matter of assigning a block to multiple destinations. The multiple assignments will infer additional logic and interconnections, splitting the original blocks fanout into local sub-trees for each FPGA if possible.

Figure 110: Replication and partitioning into both FPGAs requires only 16 extra pins



Because we are making two copies of our new block, the end use of resources will increase, although maybe not as much as first anticipated. Part of the replicated block placed in the second FPGA may not be required to drive logic there while that part of the original is already driving the logic in the first FPGA. Hence parts of the replicant and original will be pruned out during synthesis.

Replication can also be used to reduce the number of IO for a high-fanout block driving multiple FPGAs. The simple, albeit unlikely scenario of an address decoder driving three FPGAs is a good illustration of how we can reduce the number of required on-board traces by using replication to create extra decoders, and then allowing the synthesis to remove unnecessary logic in each FPGA.

Replication is such a helpful trick that, when we are partitioning, or preferably before, we should be on the look-out for replication opportunities to lower the IO requirement.

Replication is also very useful for distributing chip support items such as clock and reset across the FPGAs, as we shall see in section 8.5.

8.2.9. Multiplex excessive FPGA interconnect

As mentioned, large SoCs with wide buses may not partition into a number of FPGAs without overflowing the available IO resources, even when using replication and other techniques. In that case we can resort to using time-division multiplexing of more than one signal onto the same trace.

Multiplexing is a large subject, so in order not to break the flow of our discussion here, we shall defer the detail until section 8.6 below.

Let us assume for the moment that and necessary signal multiplexing has been added and that the interconnect between the FPGAs has been defined. We are now ready to fix the FPGA pin locations. We do this directly by assigning signals to traces.

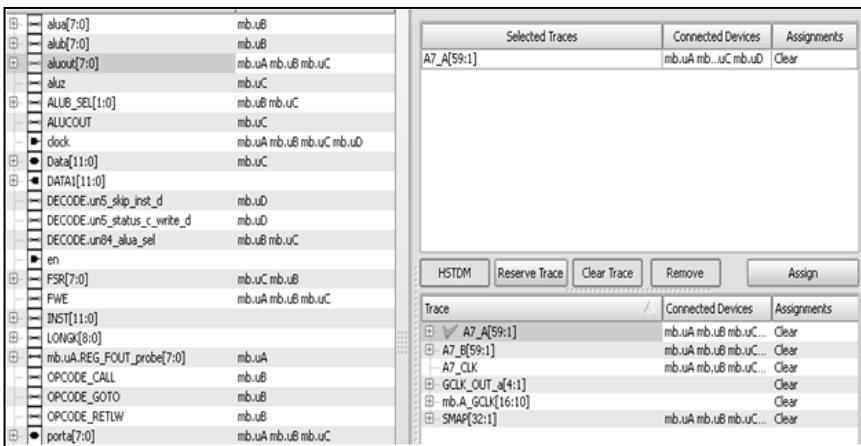
8.2.10. Assign traces

We have reached the stage in our project where we have partitioned all the SoC into FPGAs, but before running synthesis and place & route, we need to fix the pinout of our FPGAs. All along we have been considering the number of FPGA IO pins as being limited. However, really we should be talking about the connections between those pins on the different FPGAs. The number of traces on the board or the width of cables and routing switches which are able to connect between those FPGA IO pins are the real limitation.

In a well-designed platform, every FPGA IO pin will be connected to something useful and accurately represented in the board description file. Then the partitioning tools will know exactly, and without error, which traces are available and to which FPGA pins they connect. If the board description is accurate then the pin assignments will be accurate as a result.

This section of the chapter is called “Assign traces,” rather than “Assign FPGA pins,” because that is what we are doing. We should not think that we are assigning the signals at the top of each FPGA to a pin on that FPGA, but rather we should think of it as assigning signals to traces so that these propagate to fix all FPGA pins to which the trace runs.

Figure 111: Certify tool's trace assignment environment



EDA tools such as Certify automate the trace assignment process while keeping track of available traces on the board and permissible voltage regions and allowing automatic signal assignment. The trace assignment environment should help us recognize clock, resets and other critical signals. It should also help us to sort traces by their destinations and fanout and filter suitable candidates for assignment of particular traces. One view of Certify's trace assignment window is shown in Figure 111, in which we can see the signals which require assignment listed on the left. Here we can see that the user has selected a bus of eight signals called aluout which, as the adjacent list shows, needs to connect to logic which has been partitioned into devices mb.uA, mb.uB and mb.uC. Once a signal or group of signals is selected, at the bottom right there appears a filtered list of candidate traces on the board i.e., those which are available and which connect to at least the required FPGAs (or other board resources should that be the case). We then pick our candidate and confirm the assignment with the relevant button.

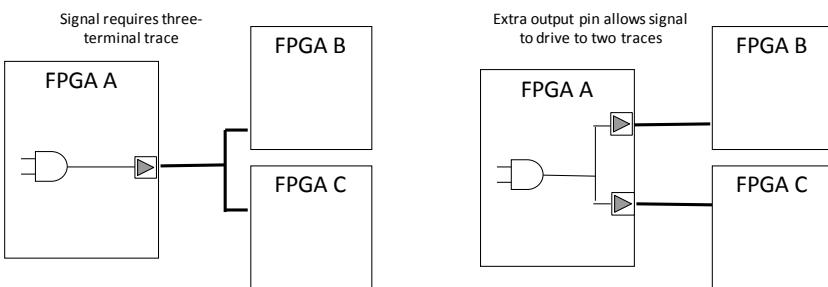
The recommended order to assign our signals to traces is as follows:

- Assign clock, reset and other global inputs if any.
- Assign test points or probe outputs to match the on-board connections.
- Assign top-level ports which connect to dedicated resources or connectors.
- Assign rest of inter-device signals, being aware of voltage-level requirements.

In each case, there will probably be more than one candidate trace for any selected signal. We choose from candidate traces in the following preference order:

- Any available trace that has the same end-points as the signal.
- If no trace meets the criteria in 1, then a trace with the same end-points as the signal, plus minimum number of superfluous end-points. Superfluous end-points mean that FPGA pins will be wasted.
- If no trace meets the criteria in 2, then we must split the signal onto two traces using two output pins at the signal's source FPGA, as seen in Figure 112.

Figure 112: Splitting signal onto two FPGA pins, using two traces to reach destination pins



When there is no trace including at least the end-points required by the signal, then multiple traces between the driving FPGA and each of the receiving FPGAs can be used.

Our EDA tools should be able to split the output onto two FPGA pins automatically without a change to the RTL. This is achieved in Certify by simply assigning the signal to more than one trace. Then the tool increments the pin count on the driving FPGA and the signal is replicated onto the extra trace, or traces.

In general, while following the above recommended preference order there may be multiple equivalent candidate traces to which a signal can be applied and in those cases it is irrelevant which one we choose. This reinforces the notion that for many signals it is not important which FPGA pins we use, as long as they are all connected together on the same trace.

8.2.10.1. Note: automatic trace assignment

Given that the above procedure is fairly methodical, it is relatively simple for EDA tools to make automated assignment following the same rules. In fact, we can take a hybrid approach by making the most important connections manually and allowing the tools to automatically complete the assignments for less critical signals.

Automatic tools will follow a similar order of assignment as we would ourselves. We can minimize the manual assignment steps by giving guidance to the automated tool when it is assigning traces from FPGA to other resources. By naming the signals and the resource pins using the same naming convention, the tools can then recognize, for example, that signal `addr1` connects to ram `addr1` and its associated trace, rather than any other candidate.

In this way we can methodically step through the trace assignments and guided by our tools, we can quickly assign thousands of FPGA pins without error.

8.2.10.2. Note: trace assignment without EDA tools

The authors are aware of at least one team that has developed its own pin assignment scripts over many projects. These scripts extract the top-level signal names from the log file of dummy FPGA synthesis runs and put them into a simple format that can be imported into the usual desktop spreadsheet applications. The team then uses the spreadsheet to text search the signal names and assigns them to correct FPGA pins. A bespoke board description which includes all the FPGA connections of note is also imported into the spreadsheet. The spreadsheet would then generate pin lists which can be exported and scripted again into the correct UCF format for place & route.

This approach is rare because not only is there the effort of developing and supporting the scripts, spreadsheet and so forth, but it may not also be the core competency of the prototyping team and the best use of its time. In addition, we are introducing another opportunity for error in the project. However, this example does underline the point that we can do FPGA-based prototyping without specialist EDA tools if we are skilled and determined. This might be a “make vs buy” decision that most would avoid, and instead default to using the commercially tried and tested partitioning tools available on the market today.

8.2.11. Iterate partitioning to improve speed and fit

Our first aim, as stated at the start of this chapter, was to organize blocks of the SoC design into FPGAs in such a way as to balance FPGA utilization and minimize the interconnect resources. Once this is achieved, we might want to step back and tweak the partition to improve it, possibly to improve performance.

Recommendation: Take a number of different approaches to partitioning, perhaps with different team members working in parallel, because the starting point can make a big difference to the final outcome. The initial block assignment in particular has such a strong impact on later partitioning decisions, so even just having a number of tries at the initial partition may bring some reward.

For example, the major blocks might be split differently and partitioned so that multiplexing is required on a different set of signals, over a different set of traces. The multiplexed paths often become the most critical in the design, so if we can perform multiplexing in a less critical part of the design, then this might raise the overall system performance.

Similarly, replicating larger blocks early in the partitioning, rather than smaller blocks later on, might yield better overall results. If we can make a collection of different coarse partitioning decisions at the start and then explore these as far along the flow as is practical then we can pick the most promising for first completion but perhaps revisiting others when we have learned from our first attempts.

It is often rewarding to discuss partitioning strategies with the SoC back-end layout team as they may have partitioning ideas to share which yield lower routing congestion.

Once we have a finished partition for taking on to the rest of the flow and downloading onto the boards, we can split our efforts and while the FPGA is being brought up and debugged in the lab, we can spare some time in parallel to explore improved results.

During the partitioning task, not only does the design need to be split into individual sub-designs but also consideration must be given to the overall system-level performance of the prototype in the lab. Let us look now at general methods for improving performance in a prototype.

8.2.11.1. Note: explore non-obvious partitions

As we explore different partition ideas, we should try to look beyond the natural functional boundaries within the design. For example, a prime reason why an acceptable partition cannot be found at first might be that a large interconnected block takes up too much FPGA resource. Looking into the next-level hierarchy we find a similar situation, but taking a different view on the blocks might show a datapath structure or a regular channel arrangement which can be split along its length rather than across its block boundaries.

Another alternative partitioning strategy might be to identify all logic within a particular clock domain and then assign it all into the same FPGA. Partitioning tools should have a scripting or graphical method for selecting all logic that is driven by a specific clock. This may also increase performance in a design because the critical path within a domain would avoid traversing an inter-FPGA path. This ideal situation may not arise often in practice, however, owing to other constraints, and in particular, the natural tendency to partition by functionality, rather than by clock domain.

8.3. Automated partitioning

We touched upon impact analysis earlier and considered that it might be possible to automate partitioning based on a similar approach.

Generally, automatic partitioning tools work towards the same prime goals as we would ourselves, namely to minimize IO connectivity between FPGAs and balance resource utilization inside the FPGAs, but they do not have the intelligence to replace an experienced prototyper in finding an optimal solution. What they can do very well, however, is to try very many strategies until something works. An ideal combination may be to use our skill and knowledge to assign an initial set of blocks and then allow the automatic tool to complete the rest.

At the very least we will need to guide the tools. Here are some tasks which should be done manually in order to assist an automatic partitioner:

- **Group pins together** that need to be connected to an off-FPGA resource (e.g., memory or external interface). If there are no constraints to keep pins together the partitioner may split the pins across all FPGAs. This would be a problem because a typical external resource like a memory is normally connected to only one FPGA.
- **Constrain resource usage per device:** the automatic partitioner may have a default, but in any case, the available resources (gates, logic, memory) inside an FPGA should be constraint to a maximum of 50 to 70%.
- **Populate black boxes or manually assign a resource count** so that even the black boxes appear to have some size and then the partitioner will reserve space for that black box. Autopartitioners cannot split black boxes.
- **Assign clocks and reset manually:** as we would for manual partitioning, special components like the clocks, resets and startup should be replicated into all FPGA and this must usually be done manually (see next section).
- **Group blocks together for peak performance:** an example here is the manual partitioning of blocks which should stay in one FPGA to get highest performance.
- **Allow automatic partitioner to perform multiplexing:** the quality of the results will vary from tool to tool, but in those designs which need IO multiplexing, the automated tool may be able to find a solution which allows a lower multiplexing ratio and hence higher system performance.

There are a number of commercially available automated partitioning tools, each with a different approach. However, we must not think of these tools as a push-button or optimal solution. The only partitioning tools which come close to this push-button ideal are aimed at quick-pass, low utilization and low performance results, best suited to emulator platforms. For FPGA-based prototyping, where high-

performance is our main aim, this kind of fully automated partitioning is not feasible and it will always be both necessary and beneficial for us humans to stay involved in the process.

8.4. Improving prototype performance

The timing for signals to travel between FPGAs is typically longer than signals that remain inside a given FPGA, therefore inter-FPGA timing is likely to become the limiting factor of the system clock rate, especially in cases where multiplexing is used. We will have a greater impact upon prototype performance if we focus on IO timing and critical inter-FPGA paths. This is done using timing constraints.

SoC top-level constraints apply to the FPGAs down to the level of the individual FFs in their respective clock domains and also between them if cross-domain timing is defined. This is particularly powerful when we can ensure that each FPGA boundary has an IO FF which aligns with a sequential element in the SoC, as we shall see in a moment.

Recommendation: after partitioning, the constraints act upon FPGA synthesis and place & route back-end on each FPGA in isolation so we also need to generate implementation constraints for each FPGA to ensure maximum performance.

We will not give details here of timing constraints in FPGA tools because the references, including the tool vendor's user guides, are the best source of such information but here are some short notes most relevant to this discussion:

- Synthesis uses estimated timing and maximum delay models for the FPGA.
- Place & route uses exact timing models for maximum timing and statistical estimates for minimum timing.
- The synthesis and place & route tools are timing driven so all paths in the FPGA are constrained, even if only by the global defaults unless explicitly given relaxed timing using false path constraints or other methods to break a timing path.
- Black boxes break a timing path so it is recommended to provide timing information for any black boxes in order to constrain any connected paths.
- FPGA IO pins are considered constrained by the applicable clock constraint.

In general, any FPGA design benefits from plentiful and accurate timing constraints but sometimes designers may not have enough understanding of the final environment or clock domain relationships to create them and this is particularly true for IO constraints. In FPGA-based prototyping we have an advantage in that we

have a good understanding of the boundary and external conditions for every FPGA pin. For example, we know the board trace performance, the exact route taken across the board and even the logic in the source FPGA which is driving the signal. These boundary conditions can be automatically translated into the timing constraints necessary for driving both synthesis and place & route for each FPGA in isolation. This process is called time budgeting.

8.4.1. Time budgeting at sequential boundaries

We can improve the timing of any inter-FPGA path by ensuring that there are FFs at the FPGA pins on the path. This is because the clock-to-output delay on the source FPGA output pin and the set-up time on the destination FPGA input pin(s) are minimized.

As we saw in chapter 3, every FPGA IO pad has multiple embedded FFs and these are available to us “free-of-charge,” so why not use them? If we can use these IO FFs in our prototype then they will also provide an additional benefit that the SoC top-level constraints will apply by default to all FFs.

FPGA-to-FPGA timing using the above ideal mapping is constrained by the top-level constraints which automatically propagate to the FFs at each end of any path, internal or external, unless overridden locally. Therefore, the constraints applying to SoC FFs mapped into IO FF are simplified and the FPGAs can be more easily constrained in isolation. This helps in our EDA tool flow because top-level constraints will be automatically reapplied to the FPGAs during each design iteration.

If the partition or the SoC design does not provide FFs which can be placed readily in the FPGA’s IO FF, then is it feasible to add these manually or via scripted netlist editing? The addition of extra FFs into an SoC path just so that they can be mapped into IO FFs would, of course, introduce pipeline delays into that path, altering its system-level scheduling. For prototyping purposes we cannot arbitrarily add such extra FFs, tempting as this may be for the improved performance, without checking with the original designers and probably adding compensating FFs elsewhere in order to maintain scheduling across the design.

Recommendation: addition of pipeline FFs can improve prototype performance but must be acceptable to the SoC design team and a re-run of system-level simulation is recommended.

It is therefore preferable to have FFs at every SoC block boundary and to only partition at those block boundaries and these are indeed recommendations of a possible project-wide Design-for-Prototyping approach that we explore further in chapter 9.

8.4.2. Time budgeting at combinatorial boundaries

If the insertion of FFs at FPGA boundaries or movement of the existing FFs to those boundaries is not possible for all signals, then careful timing constraints for the combinatorial paths across the FPGA boundaries must take place. In this case, we need to evaluate and divide the timing budget between the FPGAs based on the complexity of each section of the path.

Considering a typical path origination in an internal FF on one FPGA that ends at an internal FF in a different FPGA, we would need to break the applicable FF-to-FF constraint, perhaps derived from the top-level SoC constraints mentioned above. Since only the total path is controlled by the system-level constraint we need to determine how much of that constraint should be applied to the two parts of the path as mapped into the two FPGAs. The resulting IO constraints would then be passed on to subsequent synthesis and place & route for each FPGA.

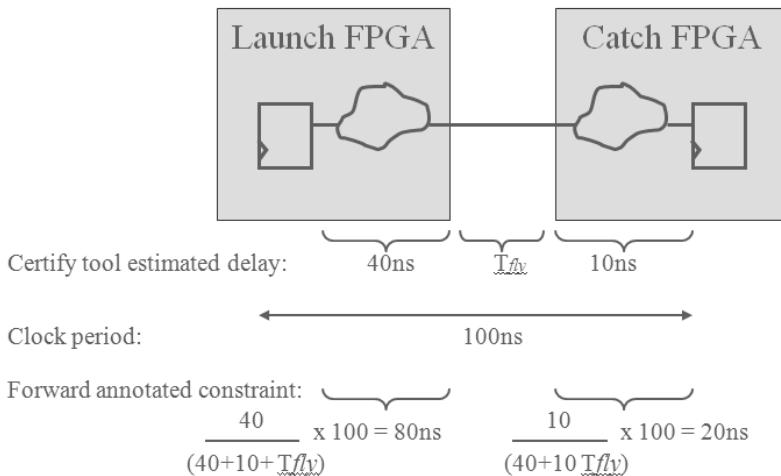
This is especially important for designs which require highest performance because the presence or absence of accurate IO constraints will drive quite different results in the place & route tool. By default, both synthesis and place & route working on the isolated FPGA after partitioning will assume that a whole period of the respective clock is available for propagating the signal to/from the IO pin to/from the internal FF. However, this assumption will almost certainly not be correct.

For example, if the signal has come from an internal FF in another FPGA, then the data will have to traverse that FPGA's internal routing, its output pad delay plus the board trace delay before arriving at the receiving FPGA's pin. The receiving FF's clock will probably have been generated internally in the receiving FPGA. We can therefore see that considerably less than the whole clock period would be available for propagating the signal through the input pad to the receiving FF in order to meet its set-up timing requirement. Relying on the default is risky and so we need to give better IO constraints at combinatorial boundaries, but what should those values be?

It is worth noting that a semi-manual approach could be taken: extracting delay information from a first-pass FPGA timing analysis and then using a spreadsheet to calculate more accurate IO constraints. However, creating IO constraints for many hundreds or even thousands of signals at combinatorial partition boundaries would be a long and potentially error-prone approach. It would also need to be repeated for every design iteration.

Another workaround would be to apply a default of a half-clock cycle of the receiving FF's clock and this coarse value may be adequate for a low performance target.

Figure 113: Time budgeting at combinatorial partition boundaries



The good news is that automatic and accurate timing budgeting at combinatorial partition boundaries is possible. The Certify tool, for example, uses a simple algorithm to budget IO constraints based on the slack of the total FF to FF. The synthesis is run in a quick-pass mode to estimate the timing of a path accounting for IO pad delays and even the trace delay. Multiple FPGA boundaries and different clock domains in a path are also incorporated in the timing calculation. The result is a slack value for every multi-FPGA path and we can see the proportion of the path delay shared between the FPGAs. An example of this is shown in Figure 113 with exaggerated numbers just to make the sums easy. We see that the timing budgeting synthesis has estimated that 40ns of the total clock constraint of 100ns is spent traversing the first FPGA and 10ns is spent in the second FPGA. There is also a time allowance for the “flight time” on the trace between the FPGAs.

The total permitted path delay (usually the clock period) is budgeted between the devices in proportion to each FPGA’s share of the total path delay. Therefore if either the launching or catching FPGA has a larger share of the total path delay, then the place & route for that FPGA will also have received a more relaxed IO timing constraint i.e., the path is given more time. This is a relatively simple process for an EDA tool to perform but can only be done if the tool has top-down knowledge of the whole path.

This all assumes an ideal clock relationship between the source and destination FPGAs on our boards and we may need to take extra steps to ensure that this is really so, as we shall see in section 8.5.1.

8.5. Design synchronization across multiple FPGAs

Our SoC design started as a single chip and will end up the same way in silicon but for now, it is spread over a number of chips and the contiguity of the design suffers as a result. We have discussed some of these ways we can compensate for the on-chip/off-chip boundary to improve performance and later we shall discuss multiplexing of signals.

There are three particular aspects of having our design spread over multiple chips that we need to concentrate upon. These are the clocks, the reset and the start-up conditions. For complete fidelity between the prototype and the final SoC, then the clock, reset and start-up should behave as if the hard inter-FPGA boundaries did not exist. Let's consider each of these in turn.

8.5.1. Multi-FPGA clock synchronization

We saw in chapter 5 how the FPGA platform can be created with clock distribution networks, delay-matched traces and PLLs in order to be as flexible as possible in implementing SoC clocks. Now is the time to make use of those features.

There are two potential problems that occur when synthesizing a clock network on a multi-FPGA design:

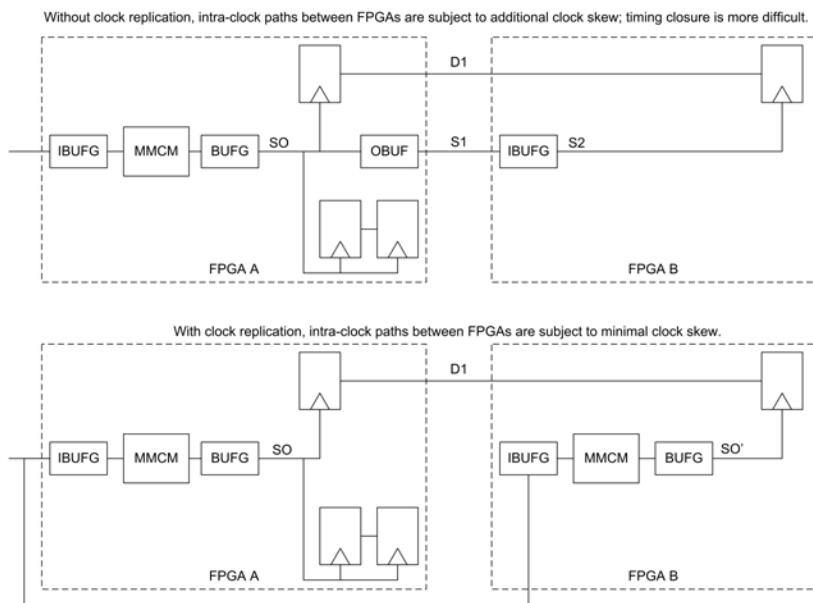
- **Clock skew and uncertainty:** a common design methodology uses the board's PLLs to generate the required clocks and distribute them as primary inputs to each FPGA. The on-board traces and buffers can introduce some uncertainty and clock skew between related clocks due to the different paths taken to arrive at the FPGAs. If ignored, this skew can cause hold-time violations on short paths between these related clocks.
- The back-end place & route tools can typically resolve hold-time violations within the FPGA but cannot currently import skew and uncertainty information that is forward-annotated from synthesis. To address this problem, the FPGAs own PLLs (part of the MCMMs in a Virtex®-6 FPGA) can be used for clock generation in combination with, or instead of the board's PLLs,. The back-end tools understand the skew and uncertainty of an MCMM and can account for them during layout. However this use of distributed MCMMs can introduce the second of our potential problems mentioned above.
- **Clock synchronization:** When related clocks are regenerated locally on each FPGA there is a potential clock-synchronization problem that does affect the original SoC design, where clocks are generated and distrusted from a common source. The problem becomes particularly apparent when multiple copies of a divide-by-N clock are derived from a base clock but

are not in sync because of reset or initial conditions and just unlucky environmental glitches.

Partitioned designs have their clock networks distributed across the FPGAs and other clock components on the board. Spanning a large number of different clocks across all FPGAs can be made easier by successful clock-gate conversion (see chapter 7), reducing the number and complexity of the clocks. However, there will still be a number of clock drivers which need to be replicated in multiple FPGAs. It is important that these replicated clocks remain in sync and that any divided clocks are generated on the correct edge of the master clock. This is dependent upon the application of reset on the same clock edge at each FPGA as we shall describe in the next section.

We therefore have a generic approach to clock distribution as shown in Figure 114,

Figure 114: Clock distribution across FPGAs



in which we see the source clock generated on a board while the local MMCMs are used in each FPGA to resynchronize and then redistribute the clocks via the BUFG global buffers. In many cases this small FPGA-clock tree will need to be inserted into the design manually. The latest partitioning tools are introducing features which automatically insert common clocking circuitry into each FPGA.

Our earlier recommendation to design the SoC with all clock management in a top-level block will really help now. We only need to make changes at that one top-level block and then use replication to partition the same structure into each FPGA. Even if the clocking in the SoC RTL is distributed throughout the design then replication will help us restrict the changes to fewer RTL files than might otherwise be necessary.

Replication of clock buffers might be avoided if we use a technique discussed in section 8.2.11.1 above for partitioning by clock domain. Success of this approach will depend on relative fanout of the different clocks and the number of paths between domains.

Whatever the partitioning strategy, each FPGA is a discrete entity and for clock synchronization, each must have its own clock generation rather than relying on clocks arriving from a generator in another FPGA. Therefore a clock generator must be instantiated in each FPGA even if only a small part of the SoC design is partitioned there.

Let's look a little more closely at the clock generator and how it helps us during prototyping. The MMCM's PLL has a minimum frequency at which it can lock and so the input clock to the FPGA must drive at least at that rate. Virtex[®]-6 MCMMs have a minimum lock frequency of 10MHz, compared to 30MHz or higher for previous technologies, which makes them particularly useful for our purposes. They are able to generate much slower clocks than they can accept as inputs. Our task is therefore to assemble a clock tree across the prototype where we maintain a higher frequency outside of the FPGAs and then divide internally while keeping the internal clocks in sync.

We achieve this as follows:

- If the SoC has clock generator at the top-level as recommended then
- Create a clock generator block in RTL to replace the equivalent part of the SoC generator. We use a global base clock to drive MCMMs which generate slower derivatives via its divide-by-n outputs.
- Select any one global system clock generated on board; its frequency must be above the minimum MMCM lock frequency.
- Drive the input to the new RTL clock tree with the global clock.
- In the new clock generator RTL, create a tree of MMCM and BUFG instantiations to create all necessary sub-clocks.
- During partitioning, replicate the MMCM and BUFGs into each FPGA as required to drive logic assigned there.
- During trace assignment, the global clock must be assigned to the global clock inputs for the FPGA.

- If clock gating and generation is more distributed then we may need to instantiate BUFG and MMCM components directly into different RTL files, but we should always be on the lookout for how replication can make this process easier.

For multi-board prototypes, we may need an extra level of hierarchy in the clock tree. We should use the on-board PLLs to drive the master clock to each board and resynchronize using PLLs at each board, using the board's local PLL output to drive each FPGA locally as described above. Skew between boards is avoided by using PLLs and matched-delay clock traces and cables as described in section 5.3.1.

Because multiple PLLs and MCMMs may be used, the local slow clocks must be synchronized to the global clock. This is achieved by using the base clock as the feedback clock input at each MMCM.

We might ask why we do not generate all clocks using the on-board global clock resources. After all, we might have placed specialist PLL devices on the board with, for example, even lower minimum lock frequencies. The issue to be aware of here is that, depending upon the board, there may be some skew between the arrival times of the clocks at each FPGA, especially for less sophisticated boards not specifically designed and laid out for this purpose. This effect will be magnified in a larger system and could lead to issues of hold-time violations on signals passing between FPGAs.

8.5.2. Multi-FPGA reset synchronization

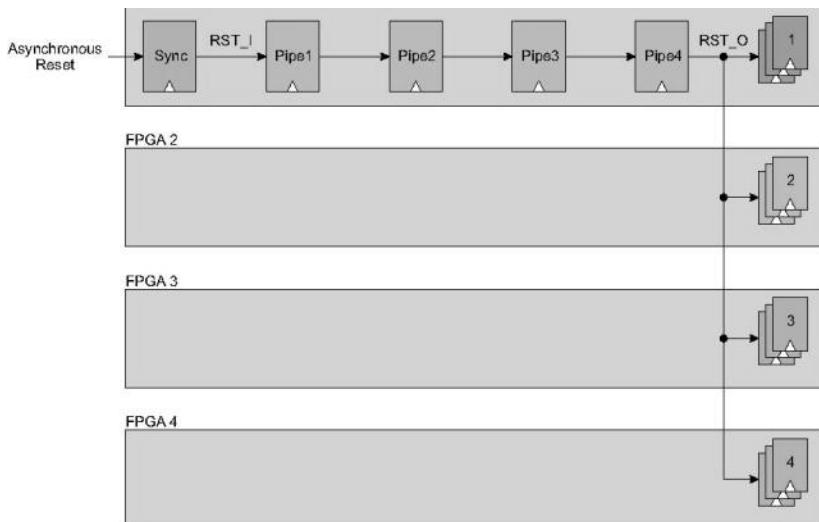
There will be “global” synchronous resets in the SoC which will fan out to very many sequential elements in the design and assert or release each of them synchronously, on the same clock edge. As those sequential elements are partitioned across multiple FPGAs it is obviously critical to ensure that each still receives reset in the same way, and at the same clock edge. This is not a trivial problem but distributing a reset signal across several FPGAs in a high-speed design can be achieved with some additional lines of code in the RTL design and a partitioning tool that allows easy replication of logic.

The enabling factor in this approach is that it is unlikely that the global reset needs to be asserted or released at a specific clock edge, as long as it is the SAME clock edge for every FPGA. Therefore, we can add as many pipeline stages into the reset signal path as we need and we shall use that to our advantage in a moment.

Another factor to consider is that the number of board traces which connect to every FPGA is often limited so the good news is that we do not need any for routing global resets. Instead we create a reset tree structure which routes through the FPGAs themselves and then uses ordinary point-to-point traces between the FPGAs.

Considering the design example in Figure 115 in which a pipelined reset drives sequential elements in four different FPGAs, it is clear that the elements in FPGA 4 will receive the reset long after those in FPGA 1.

Figure 115: Pipelined synchronous reset driving elements in 4 FPGAs

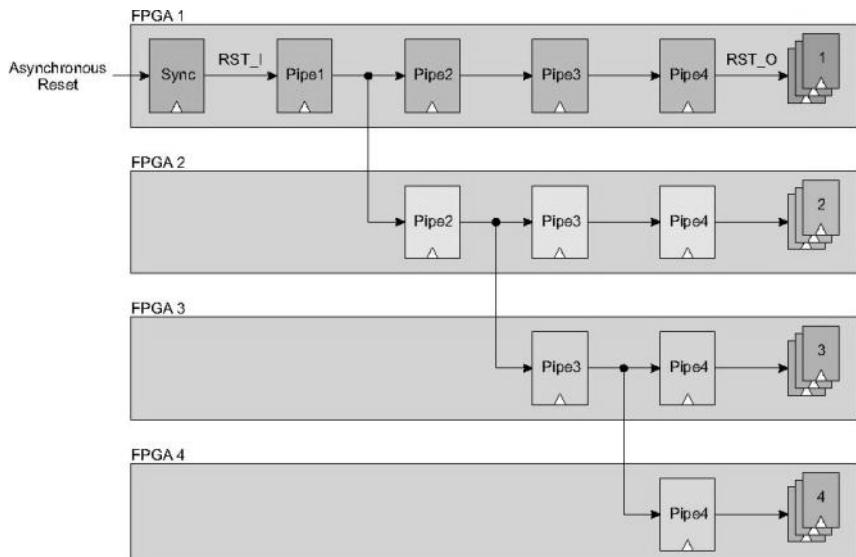


Notice also that the input could be an asynchronous reset, so the first stage acts as a synchronizer (with double clocking if necessary for avoiding metastability issues). Routing through the FPGAs in this way is not acceptable except for very low clock rates, so to overcome this we will replicate part of the pipeline in each of the FPGAs as shown in Figure 117.

Readers using Certify will find further instructions on the correct order for replication in the apnote listed in this book's references.

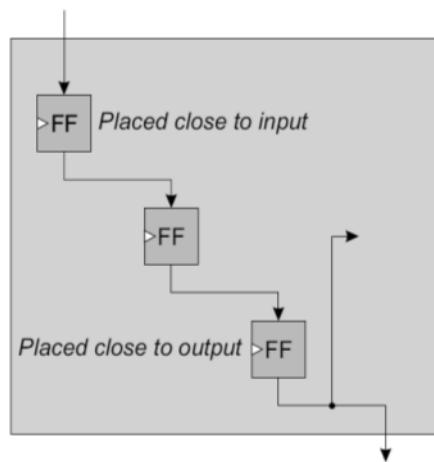
The first stage is not replicated because the synchronization of the incoming reset signal has to be done in only one place. There is still a possibility that the pipeline stage in each FPGA would introduce delay because if there is one FF in each stage, then it might be placed near the input pad, the output pad or anywhere in between; this might also be different in each FPGA. The answer is to use three FFs for each pipeline stage, as shown in Figure 116.

Figure 117: Tree pipeline created by logic replication



This allows the first and third FF to be placed in an IO FF at the FPGA's edge. Then there is a whole clock period for the reset to propagate to the internal FF and on to the output FF, greatly relaxing the timing constraint on place & route. Once again, these pipe stages only introduce an insignificant delay compared to the effect of the global reset signal itself.

Figure 116: A three-FF pipeline stage gives more freedom to place & route



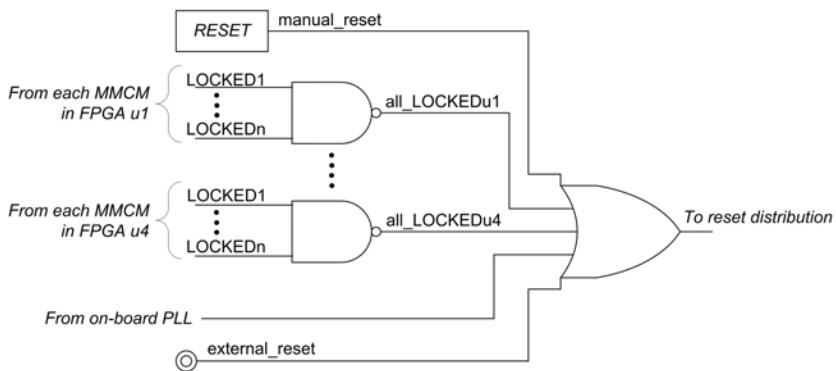
Note: the synthesis might try to map the pipeline FFs into a single shift register LUT (SRL) feature in the FPGA, which would defeat the object of the exercise so the relevant synthesis directive may be required to control the FF mapping. In the case of Synopsys FPGA synthesis, this would be `syn_srlstyle` and for good measure `syn_useioff` would be used to force the two FFs into the IO FF blocks, although that is the default.

8.5.3. Multi FPGA start-up synchronization

Using the above technique we can ensure that all FPGAs emerge from start-up simultaneously. But this is of little use if the clocks within a given FPGA are not running correctly at that time. We must also ensure that all FPGA primary clocks are running before reset is released. This is important because, owing to analog effects, not all clock generation modules (MMCMs, PLLs) may be locked and ready at the same time. We must therefore build in a reset condition tree. This can be accomplished by adding a small circuit like the one shown in Figure 118 which would be distributed across the FPGAs.

Here we see a NAND function in each FPGA to gate the LOCKED signals from only those MMCMs which are active in our design. This would be a combinatorial function or otherwise registered only by a free-running clock that is independent of the MMCM's outputs.

Figure 118: Example reset condition tree



Each FPGA then feeds its combined `all_LOCKED` signal to the master FPGA in which it is ORed to drive the reset distribution tree described in the previous section. The locked signals of any on-board PLLs used in this prototype must also be gated into the master reset and there may other conditions not related to clocks

which also have to be true before reset can be released, for example, a signal that external instrumentation is ready and, of course, user's "push-button" reset should also be included. Our example shows that these are all active high but of course the reset gate can handle any combination. The whole tree would be written in RTL which is added into the FPGA's version of the chip support block at the top level and we can use replication to simplify its partitioning.

The global reset is only released when all system-wide ready conditions are satisfied. The reset will also then release all clock dividers in the various FPGAs on the same clock edge so that all divide-by-n clocks will be in sync across all FPGAs.

8.6. More about multiplexing

The partitioning is done. The resource utilization of all FPGAs is well balanced and within the suggested range. Furthermore, the design IOs per FPGA are minimized but after such good work, there is still a chance that there are not enough FPGA pins available to connect all design IOs, or to be more accurate, there are not enough on-board traces between some of the FPGAs. As mentioned in section 8.2.9 above, the solution is to multiplex design signals between FPGAs in question. Multiplexing means that multiple compatible design signals are assembled and serialized through the same board trace and then de-multiplexed at the receiving FPGA. We shall now go into more detail on how this is done and also compare different multiplexing schemes. We shall also explain the criteria for selecting compatible signals for multiplexing and give guidance on timing constraints.

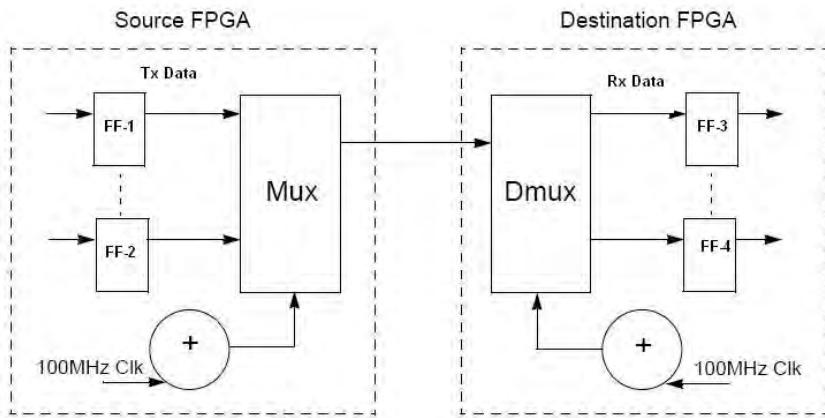
8.6.1. What do we need for inter-FPGA multiplexing?

To multiplex signals between FPGAs we need a number of elements including a multiplexer (mux), demultiplexer (dmux), clock source and a method for keeping all of these mutually synchronous. All of these elements are seen in the in Figure 119 (see next section for full explanation).

If we have freedom to alter our RTL then theoretically, these elements could be manually added at each FPGA boundary. We would need to add the multiplexing elements after partitioning or add the elements into the RTL from the start, therefore pre-supposing the locations for partition boundaries. In both cases, the rest of the SoC team might see this as stepping too far away from the original SoC RTL and introducing too many opportunities for error.

Most teams would not contemplate such widespread changes to the SoC RTL and instead they rely on automated ways to add the multiplexing, either by scripted direct editing of the post-synthesis netlists or by inference during synthesis, based on direction given by the partitioning process. We shall explain more about this in a moment.

Figure 119: Basic time-division multiplexing of inter-FPGA signals



Whatever the method for introducing the muxing, the basic requirement of the scheme is to transfer IO data values from one FPGA to another within one design clock. To achieve this, the serial transfer clock (also named multiplexing clock or fast clock) must sample those data values faster than the design clock to guarantee that all the data is available in the receiving FPGA before the next active design clock edge.

As an example, let's assume that we have four IO data values to transfer between two FPGAs which are multiplexed on a single on-board connection i.e., a mux ratio of 4:1. If this part of the design is running at 20MHz then, to transfer the four design IOs within a design clock cycle, we need a transfer clock which is at least four times faster than the design clock. Therefore the transfer clock must be 80MHz at minimum. In practice it needs to more than four times faster for a multiplexing of 4:1 because we need to ensure that we meet the setup and hold times between the data arriving on the transfer clock and then being latched into the downstream logic on the design clock.

In most of the cases where multiplexing is used, it decreases the overall speed of the design and is often the governing factor on overall system speed. The serial transfer speed is limited by the maximum speed through the FPGA IOs and the flight time through the on-board traces. Therefore, with these physical limits the multiplexing scheme needs to be optimized to allow the prototype to be run at maximum speed.

Multiplexing is typically supported by partitioning tools which insert the mux and dmux elements and populate them with suitable signals. For example, in the Certify tool there are two different types of scheme called certify pin multiplier (CPM) or high speed time domain multiplexing (HSTDm).

Based on the relationship of the transfer clock and design clock, we can differentiate between two types of multiplexing. Asynchronous multiplexing, where the transfer clock has no phase relation to the design clock, and synchronous multiplexing, where the transfer clock is phase aligned to the design clock, and probably even derived from it.

8.7. Multiplexing schemes

The partitioning tools insert multiplexing based on built-in proprietary multiplexing IP elements. We normally do not know, or probably need to know, every detail of these elements but we shall explore some typical implementations in the rest of this section. At the end of the section a comparison of the different schemes is shown.

8.7.1. Schemes based on multiplexer

The simplest scheme is a mux in the sending FPGA and a dmux on the receiving FPGA, much as we saw in Figure 119 but with the source FFs omitted. In the source FPGA there is a 100MHz transfer clock which drives a small two-bit counter which cycles through the select values for the mux. Every 10ns a fresh data value starts to traverse the mux, over the board trace and into the dmux in time to be clocked into the correct destination FF. Meanwhile, the dmux is switched in sync with the mux as both are driven by a common clock, or more likely two locally generated clocks which are synchronized as mentioned in section 8.5.3 above.

On each rising edge of the transfer clock, new data ripples through to the mux output and propagates across the trace from the source FPGA to the destination FPGA in time for the receiving register to capture it. This is an extra FF driven by the transfer clock rather than the design FF that would have captured the signal if no multiplexing scheme had been in place.

To use this simple scheme, we need to select candidate signals that will propagate across the mux and dmux in order to meet the set-up timing requirement of the receiving FF. If the timing of the direct inter-FPGA (i.e., non-multiplexed) path was already difficult for the receiving FF to meet, then this is not a good candidate for multiplexing. In normal usage, we would select signals with a good positive slack and these can be estimated after a trial non-partitioned synthesis.

Best candidate IOs for this simplest kind of multiplexing scheme are those directly driven by a FF, which normally could map into IO FFs in the FPGA if there were

enough pins available. These would have the maximum proportion of the clock period available to propagate to the destination, assuming that the transfer clock is in synchronization with the design clock driving those FFs. Using IO FFs, the timing of inter-FPGA connections is more predictable and generally faster. Therefore a multiplexing scheme should use IO FFs if possible and we should use synthesis attributes to ensure that a boundary FF is mapped into an IO FF if physically possible.

Another multiplexing scheme, which is very similar to the one described above, has additional sampling FFs in the source FPGA driven by the transfer clock exactly as seen in Figure 119. Now the whole mux-dmux arrangement is in sync with the transfer clock and it is easier to guarantee timing. In fact we need not even have a transfer clock that is synchronous with the design clock but double-clocking synchronizers may be necessary to avoid metastability issues.

8.7.2. Note: qualification criteria for multiplexing nets

There are different types of signal in the SoC design, on different kinds of FPGA interconnections. Some types are suitable for multiplexing and other types should not be multiplexed. Table 19 summarizes these different types and their suitability for multiplexing.

To get the highest performance in case of multiplexing the user should carefully select the FPGA interconnection which should be multiplexed. For designs with different design clocks the user should multiplex signals coming from a low-speed clock with a higher ratio than signals from a high-speed clock. This keeps the performance of the design high.

Table 19: Which nets are suitable candidates for multiplexing?

Net topology	Suitable?	Notes
Nets starting and ending at sequential element.	YES	If possible put sequential element in IO FF.
Nets starting at combinational element and but ending on sequential element.	YES	It could be difficult to meet the timing between design clock and multiplexing clock.
Nets between FPGAs which are also top-level design IO ports.	NO	Connected external hardware cannot perform dmux.
Nets starting and ending at sequential element but in different clock domains.	NO	This has unpredictable behavior owing to setup and hold.
Nets starting and ending at sequential elements but feeding through* intermediate FPGA.	NO/(YES)	In theory it's possible to multiplex such a net but it decreases the design performance and timing is hard to estimate.
Nets crucial to the clocking, reset, start-up and synchronization of the prototype.	NO	Crucial nets should be assigned to traces before mux population is decided.

* feed-through means that there is a path through an FPGA without sequential elements.

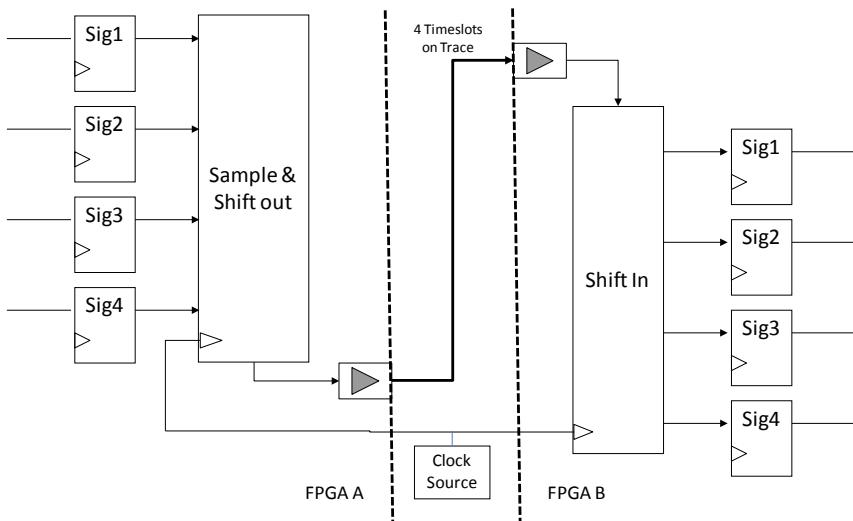
8.7.3. Schemes based on shift-registers

Another solution is based on shift-registers, as seen in Figure 120. Here the data from the design is loaded in parallel into the shift-register on the rising edge of the transfer clock and shifted out with the same clock.

In the receiving FPGA, a shift-register samples the incoming data on the transfer clock and provides the data in parallel to the design. The first sample (in this case sig 4) is available at the shift register output from the sample clock edge but an extra edge of the transfer clock may be necessary in some versions of this scheme in order to latch in the data after it has been fully shifted into the destination registers for finally clocking into the design FFs in the destination FPGA. Once again, the sending and receiving shifters need to start up and then remain in sync.

This type of scheme is well suited for boards with longer than average flight time on the inter-FPGA traces because there is no extra combinatorial delay in the path and we obtain maximum use of the transfer clock period. In particular, it is not acceptable to have new data being sampled onto the trace if the previous sample has not yet been clocked into the receiving logic. In some lab situations we can get lucky, but in others, the transmission line characteristic of the trace or a slight discontinuity in a connection can make the transfer unreliable. Therefore we have

Figure 120: Basic time-division multiplexing based on shift registers



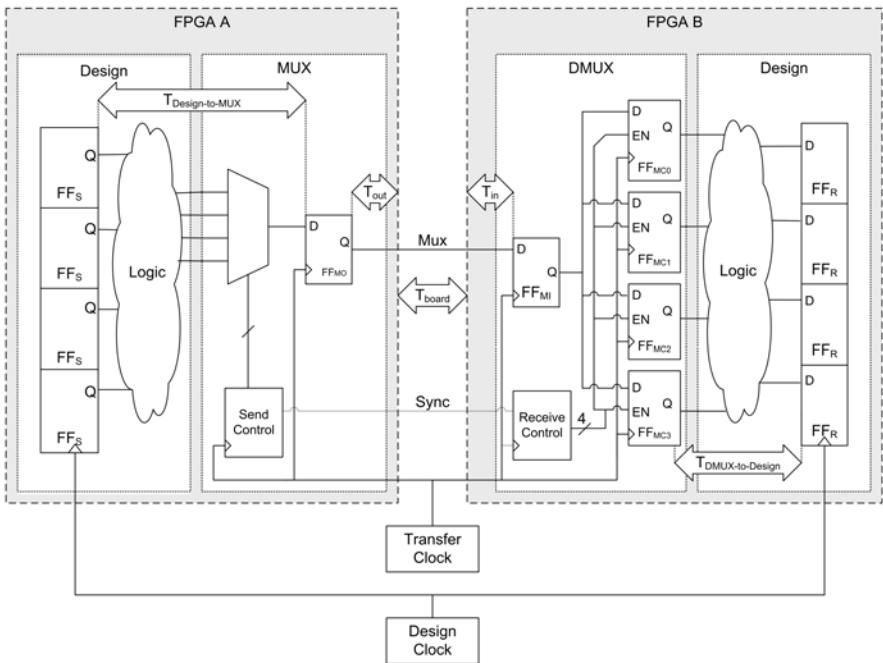
an upper physical limit on transfer clock speed and if we hit that then the only way to increase multiplexing ratio further is by reducing the overall system speed. Having done so, then even when using mux ratios of 10:1 or higher, we just have to operate the prototype at a lower clock rate.

8.7.4. Worked example of multiplexing

It is important to understand mux timing so here is a worked example of a multiplexing solution that uses a mux followed by a sampling FF.

Considering the Figure 121, in FPGA A we have a design with some flip-flops which we call FF_s followed in this example by some combinatorial design logic (in some other examples it might also be possible that there are only design flip-flops). The values are fed to the send stage which contains a multiplexer which selects each design signal in turn and an output FF, which we shall call FF_{MO} . FF_{MO} could be placed into an IO FF of FPGA A in order to improve the output timing as mentioned above.

Figure 121: Example time-division multiplexing based on sample register



Between the two FPGAs we use a single-ended connection for the multiplexed signals (shown as mux on our diagram). As mentioned above, to guarantee signal integrity we must ensure that a multiplexed sample is received and latched into the destination FF within one transfer clock cycle between the sending FF_{MO} and the receiving FF, which we shall call FF_{MI} .

As the multiplexed samples are clocked one by one into FF_{MI} in FPGA B, they are stored in a bank of capture FFs, which we call FF_{MC} . These capture FFs ensure that the samples are all stable before the next clock edge of the design clock. Our example also shows some combinatorial design logic in the receive side (but again, this might only be design FFs). The aim is to show that TDM can be used on a wide variety of candidate signals.

Now let's see how we can calculate the maximum transfer frequency and the ratio between transfer clock and design clock.

Our constraint is to transfer a data value within one period of the transfer clock cycle between FF_{MO} and FF_{MI}. The delays on the path are as follows:

- The delay through the output buffer of FPGA A (T_{out})
- The delay on board (T_{board})
- The delay of the input buffer at FPGA B (T_{in})

The maximum delay on the multiplexing connection is therefore:

$$T_{MUXmax} = T_{out} + T_{board} + T_{in}$$

If we assume typical values of $T_{out} = 5\text{ns}$, $T_{board} = 2\text{ns}$ and $T_{in} = 1\text{ ns}$ we get a maximum delay of:

$$T_{MUXmax} = 5\text{ns} + 2\text{ns} + 1\text{ns} = 8\text{ns}$$

This sets the upper limit on transfer clock frequency of:

$$F_{MUXmax} = \frac{1}{T_{MUXmax}} = \frac{1}{8\text{ ns}} = 125\text{MHz}$$

That's the theoretical rate that signals can pass between the FPGAs, but we should also respect that we are using single-ended signaling and that there can be some clock uncertainty, or jitter, between the FPGAs and we should also give some room for tolerances. Therefore, based on our experiences, we should add $T_{tolerances} = 1 - 2\text{ns}$ to provide some safety margin, depending how confident we are in the quality of the clock distribution on our boards. For our example let's assume that $T_{tolerances} = 2\text{ns}$. This results in the following calculation:

$$T_{MUXmax} = T_{out} + T_{board} + T_{in} + T_{tolerances}$$

If we assume the same values as above for the other delays we get:

$$T_{MUXmax} = 5\text{ns} + 2\text{ns} + 1\text{ns} + 2\text{ns} = 10\text{ns}$$

and a maximum transfer clock frequency of:

$$F_{MUXmax} = \frac{1}{T_{MUXmax}} = \frac{1}{10\text{ ns}} = 100\text{MHz}$$

The maximum clock frequency of 100MHz (or period of 10ns) must be given as a constraint on the transfer clock during FPGA synthesis and place & route.

Let's now consider a little more closely how the mux and dmux components are working in order to calculate the ratio between transfer clock and design clock. We have to consider two possible use cases. The first case is that that the transfer clock and the design clock are mutually synchronous i.e., they are derived from one clock source and they are phase aligned. The second case is that the transfer clock and the design clock are asynchronous, in which case, we don't know on which transfer clock cycle the transfer of the data values starts and we have to set the right constraints to make sure that it works.

Starting with the synchronous case, we see from the block diagram that there are some transfer clock cycles required to bring the data from the sending design FFs through the multiplexing registers FF_{MO} , FF_{MI} , FF_{MC} to the receiving design register FF_R .

In addition, even though the two clocks are synchronous, we have to respect the delays between the design clock and the transfer clock on the sending and receiving side. These delays are marked in the block diagram with $T_{\text{design-to-mux}}$ for the sending side and $T_{\text{dmux-to-design}}$ for the receiving side. For the following calculation of the clock ratios we assume that these delays have a constant value and we have to give these assumptions as constraints to synthesis and place & route. For our example here we shall assume that these delays are a maximum of one transfer clock cycle, which is extreme, and a maximum of 10ns.

Table 20 shows how the design signals are transferred through the multiplexing based on our assumptions above.

Consider new data DA, which is valid in the design registers FF_S . One transfer clock cycle later, the first bit, DA1, is captured into FF_{MO} . This is using our assumption that the delay between the design and transfer clocks is a maximum one transfer clock cycle and that the clocks are phase synchronous.

Table 20: How data is transferred during multiplexing

Clock cycle design/ transfer	FFS	FFMO	FFMI	FFMC0	FFMC1	FFMC2	FFMC3	FFR
1 / 1	DA	X	X	X	X	X	X	-
1 / 2	DA	DA1	X	X	X	X	X	X
1 / 3	DA	DA2	DA1	X	X	X	X	X
1 / 4	DA	DA3	DA2	DA1	X	X	X	X
1 / 5	DA	DA4	DA3	DA1	DA2	X	X	X
1 / 6	DA	DA1	DA4	DA1	DA2	DA3	X	X
1 / 7	DA	DA2	DA1	DA1	DA2	DA3	DA4	X
2 / 1	DB	DA3	DA2	DA1	DA2	DA3	DA4	DA
2 / 2	DB	DB4	DA3	DA1	DA2	DA3	DA4	DA
2 / 3	DB	DB1	DB4	DA1	DA2	DA3	DA4	DA
2 / 4	DB	DB2	DB1	DA1	DA2	DA3	DB4	DA
2 / 5	DB	DB3	DB2	DB1	DA2	DA3	DB4	DA
2 / 6	DB	DB4	DB3	DB1	DB2	DA3	DB4	DA
2 / 7	DB	DB1	DB4	DB1	DB2	DB3	DB4	DA
3 / 1	DC	DB2	DB1	DB1	DB2	DB3	DB4	DB
3 / 2	DC	DC3	DB2	DB1	DB2	DB3	DB4	DB
3 / 3	DC	DC4	DC3	DB1	DB2	DB3	DB4	DB
3 / 4	DC	DC1	DC4	DB1	DB2	DC3	DB4	DB

The shaded entries in the table show how the captured data bit is transported through the mux and dmux and is clocked into the receiving FF_R . We have highlighted in capitals where each FF in the chain has new data.

As we can see from the first column of our tab, the ratio between the design clock and transfer clock is seven, which means that the design clock has to be seven times slower than the transfer clock to guarantee correct operation.

Now it is trivial to calculate the maximum design clock frequency for our synchronous multiplexing example:

$$F_{DESIGNmaxSYNC} = \frac{F_{MUXmax}}{RATIO} = \frac{100MHz}{7} = 14.28MHz$$

So for this design, which multiplexes signals using a 4:1 mux ratio at 100MHz, we can run our design at over 14.28 MHz worst case, not the 25MHz that we might have guessed from the 4:1 ratio.

We have now seen the case where the transfer clock and the design clock are synchronous but let us consider the difference in an asynchronous multiplexing scheme where the design clock and the transfer clock are not phase aligned. The maximum transfer clock frequency is the same but we don't know the skew between the active edges of the design and transfer clock. Therefore we have to add additional synchronization time on the send and receive sides to guarantee that we meet set-up and hold time between design and transfer clocks. This adds an additional transfer clock cycle on both send and receive sides. The calculation of the maximum design frequency for the asynchronous multiplexing is:

$$F_{DESIGNmaxASYNC} = \frac{F_{MUXmax}}{RATIO + 2} = \frac{100MHz}{7 + 2} = 11.11MHz$$

We can see that the asynchronous case runs with a lower design frequency but the advantage of asynchronous multiplexing is that we don't need to synchronize the design and transfer clock and we have greater freedom from where to source the transfer clock.

To summarize the example, the important things we should keep in mind to constrain a design with multiplexing are:

- Calculate the correct maximum frequency of the transfer clock based on FPGA IO technologies, delays on board and tolerances.
- Calculate the correct ratio between design and transfer clock, respecting the difference between synchronous or asynchronous multiplexing.
- Give correct constraints in synthesis and FPGA place & route for the transfer clock and the design clock.
- Give correct clock-to-clock constraints in synthesis and FPGA place & route for the delay between transfer clock and design clock.

Having considered different kinds of multiplexing using normal single-ended signaling between the FPGAs, what can we do to have higher mux ratios but still maintain a high system speed? The answer lies in raising the maximum transfer

clock speed, using the FPGA's built-in serial IP and a more robust signaling technology.

8.7.5. Scheme based on LVDS and IOSERDES

We can further improve the transfer of data by sending the clock on a parallel path to the data. This method of sending clock and data together is called a source-synchronous interface. This makes it easier to meet timing because the clock has the same off-chip/on-chip skew as the data, especially if a well-designed board is used upon which there are matched delay traces with the same flight time.

Having the clock travel from the source, rather than be generated locally and kept in sync, would not work for driving partitioned logic in general but it is very useful for unidirectional serial data transfer.

Trace flight time works in our favor but it still places a physical limit on the maximum system speed and the way to overcome this is to use differential signaling between the FPGAs.

In this advanced case, we use the FPGA's built-in support for LVDS, which can increase transfer rates up to 1GHz. This allows much higher mux ratios without having to reduce the overall prototype clock speed. Certify's HSTDm scheme, briefly mentioned earlier, supports LVDS signaling but also uses another of the FPGA's built-in resources, the IOSERDES blocks.

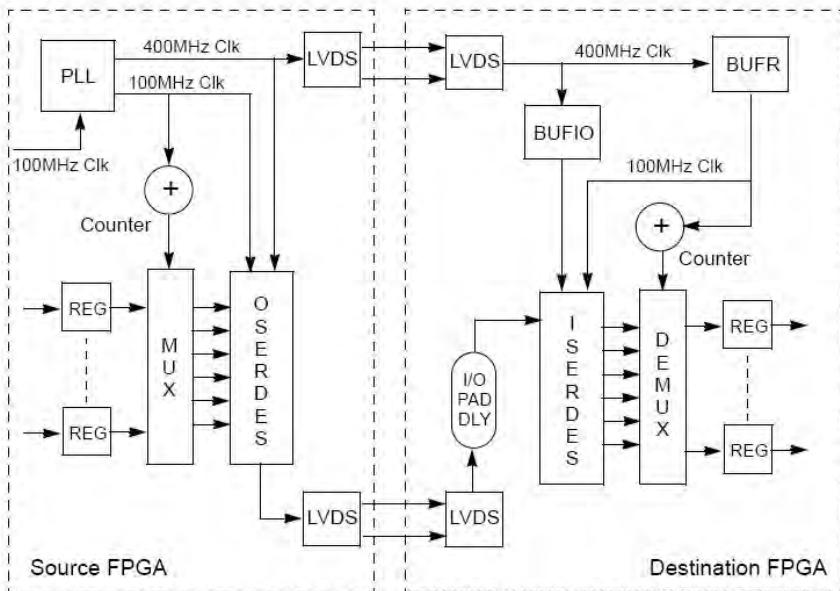
As introduced in chapter 3, modern FPGAs have dedicated serial-to-parallel converters called IOSERDES which have specific clocking and logic features for implementing high-speed serial transfers. Using IOSERDES avoids the timing and layout complexities of implementing the similar functionality within the FPGA fabric.

We can also double our transmitted data rate by using both edges of the transfer clock using the FPGA's built-in support for double data rate (DDR) operation of the IOSERDES blocks. Figure 122 gives an overview of the implementation of a high-speed TDM scheme which combines all of these advanced built-in features of a Virtex-6 FPGA.

LVDS guarantees the highest transfer speed on-board but the slight disadvantage is that there are two pins required for each serialized data stream. So, while a single-ended multiplexing scheme with a ratio of 8:1 needs only one inter-FPGA trace to transfer eight data signals, a differential IO standard needs two, so we call this a mux ratio of 8:2. Therefore the differential solution reduces the interconnections only by factor four and not by factor eight. However, when the significantly higher speed of LVDS is taken into account then multiplexing ratios of 128:2 can be considered, which gives a far greater data transfer bandwidth than is possible with a single-ended scheme.

As shown in the diagram, this IOSERDES is being run at a transfer rate of 400MHz

Figure 122: Advanced multiplexing scheme using LVDS and IOSERDES



and needs two clocks for operation. The clocks are generated from a PLL within the source FPGA based on a 100MHz clock arriving from an external source. The OSERDES is configured as an 8:1 ratio but because it operates in DDR mode, we need only have a transfer clock of 400MHz rather than 800MHz. This means that the 400MHz clock transfers 8 bits during one 100MHz period. To provide multiplexing ratios higher than eight we can use an additional mux at the input of the OSERDES but this only needs to work at the slower internal clock rate.

The data and the transfer clock are both passed to the destination FPGA via LVDS signaling but at the destination FPGA only the data goes through an IODELAY element to adjust its timing relative to the clock. The data and the transfer clock then drive the ISERDES block. Received clock is passed through BUFR block to divide it by four to create the local version of the 100MHz, which is also required for the ISERDES and the dmux control.

The diagram gives only an overview and implementations may be more complex and include so-called link training of all HSTDm multiplexing channels in order to guarantee optimal alignment between clock and data. It is important that user data is not transferred over the link while training occurs and it would be lost in any case, so only after the link training is complete is a ready signal generated. This should be

included as another input to the reset condition tree discussed in section 8.5 above. If such a training is not implemented the user has to manually adjust the IDELAY components to achieve highest speed and to guarantee the right operation.

8.7.6. Which multiplexing scheme is best for our design?

Given the number of options, it may be difficult to decide which multiplexing scheme is the best to use in our prototyping project. To help us, Table 21 shows a comparison of the multiplexing schemes described in the above sections.

Table 21: High-level comparison between multiplexing schemes

	Multiplexer	Shift register	IOSERDES
Performance	LOW – the multiplexer solution can't guarantee IO FF and this limits the speed.	MEDIUM – the shift register implemented in the FPGA fabric can limit the speed.	HIGH – best usage of FPGA resources to implement serialization.
Effort timing constraining	IO delay constraints are required for the multiplexing nets. Furthermore it's difficult to meet the constraints based on missing IO FF.	There are only clock constraints required for the multiplexing clock and constraints for design to multiplexing clock delays.	There are only clock constraints required for the multiplexing clock and constraints for design to multiplexing clock delays.
Complexity	LOW – only a simple multiplexer and some control logic is required.	LOW – only a simple shift register and some control logic is required.	HIGH – The multiplexing IP has a high complexity but it should be not a problem if provided by the partitioning tool.
Usability	On all prototyping platforms.	On all prototyping platforms.	Only usable on boards which support high-speed serial connections (differential IOs are required!)

The comparisons are shaded with the least preferable under each criterion shaded darker. As we can see in the comparison, the shift register solution is a flexible, easy to handle solution and can be used on all prototyping platforms. However, let's not

lose sight of our goal of enabling the highest transfer speed so that the prototype speed is not compromised too much. Therefore the IOSERDES solution may be offer a greater reward, but the effort may be higher and the board/system needs to support LVDS signaling across matched-delay traces.

8.8. Timing constraints for multiplexing schemes

If multiplexing is used then we need to add timing constraints for the mux and interconnects so that the implementation tools will consider them along with those for the rest of the design.

The most obvious constraint is the clock constraint for the mux clock. Based on the multiplexing scheme used and the required performance of the prototyping system, it should be possible to analyze the maximum possible transfer clock speed and then work backwards to calculate the optimum constraint accordingly. Some partitioning tools can perform timing estimation if they can be given information about the board trace delays. The derived clock constraint for the transfer clock can then be used during synthesis and place & route.

We can calculate the maximum design speed based on the maximum transfer clock speed. Remember that the mux transfers exactly one set of its inputs to the destination FPGA within a design clock period but the ratio of mux clock to design clock depends not only on the multiplexing ratio but also on the multiplexing scheme used and any skew and jitter margin added between design clock and transfer clock. The clock-to-clock ratio, rather than just the mux ratio, should be provided by the designer of the multiplexing scheme in use.

Other required constraints are the maximum delay constraints between the design clock domain and the transfer clock domain. As we have seen before, the ratio between design clock and transfer clock depends on the delay between design and the multiplexing components. If we assume that there is a defined delay like 10ns we have to give a constraint to the place & route tools to ensure that all multiplexed connections are within this range. This kind of constraint is often missing but it is important to guarantee full functionality. See the worked example above for how to calculate the correct constraint.

If we perform multiplexing correctly, all timing will be met, a high performance will be achieved and the end user of the prototype should not even be aware that multiplexing has been used.

8.9. Partitioning and reconnection: summary

For many FPGA-based prototyping teams, the first law of prototyping, that “SoC’s are larger than FPGAs,” drives the whole project. As we have seen, if a design has not been intended at all for prototyping and has a large and complex internal structure, then partitioning it into our board’s FPGA resources can present quite a challenge, and if we cannot partition our design then the project will fail. However, even with such pathologically FPGA-hostile designs, a way for it to be partitioned onto a board can be found with the question only being a matter of time allowed versus how much performance is required.

By following an FPMM-based approach and the steps given in this chapter, we give ourselves every chance of not only fitting the design onto the board but also of running it at maximum speed, even if we need to use multiplexing between the FPGAs.

There is an old joke about a lost driver stopping to ask a local pedestrian how to find his way to London, to which the local replied, *“Oh, you don’t want to start from here!”*

We would all prefer a more predictable partitioning outcome so we can make an earlier delivery of our working prototypes to the end users. Therefore we don’t want to start from an SoC design that is complex, interconnected with multiple wide buses and many inter-dependent functions. But how do we get these better starting places? How can we make prototyping more productive?

What if prototypers could take part in more of the upstream decisions of the SoC project team and guide the design towards something less hostile to FPGA, thus breaking the third law of prototyping at its source? That is the aim of the discussion in chapter 9, Design-for-Prototyping, where we consider some small procedural and technical changes in the SoC flow which will bring benefits to the whole team, not just to us prototypers. . . honest!

The authors gratefully acknowledge significant contribution to this chapter from
Pradeep Gothi of Synopsys, Bangalore

Having spent the previous chapters covering ways to account for the three laws of prototyping to configure an SoC design for a prototype, we shall now discuss ways that we might have avoided a lot of that work in the first place.

Some of the “Design-for-Prototyping” ideas in this chapter are purely technical in scope, from general and specific styles for writing our original SoC RTL up to architecture-level decisions, all with the intent of making the design more “FPGA-friendly.” Other ideas are more organizational or procedural and will hopefully provoke useful discussion amongst verification and implementations teams alike.

By employing these Design-for-Prototyping guidelines, the prototype will be easier and its verification benefits derived sooner. In effect, we are trying to obsolete the third law of prototyping, i.e., that SoC designs are FPGA-hostile.

9.1. What is Design-for-Prototyping?

Design-for-Prototyping is the art of considering the prototype early in the design process, and establishing procedural and design specific techniques to ease the development of the prototype in conjunction with all other program goals.

There’s an old expression which says that “if a job is worth doing, it is worth doing well.” If we are to employ FPGA-based prototyping in our SoC projects then we should help the prototypers to do the best job possible. We can do that by making some small changes in the SoC design style and overall project procedures, all of which are at least low impact on the rest of the SoC design flow and in fact, many are good design practice for the SoC anyway.

It should be clear from the earlier discussions that putting an SoC design into a prototyping system requires some planning in order to best accomplish the prototyping goals with minimal effort. The aim of this chapter is to help collect many of the recommendations mentioned throughout this book into a single manifesto on how we might make the prototyping tasks simpler and the project more successful. As a result, newcomers to FPGA-based prototyping will better understand how to modify their design style and development procedures to incorporate this methodology. This chapter will present some guiding principles for that change; perhaps best described as a “manifesto for Design-for-Prototyping”.

The decision to add FPGA-based prototyping to the chip design flow should be seen as a change from previous practices rather than a “bolt-on” parallel step. We should

expect some impact to all groups in the design team, including SoC designers, verification engineers, software development teams and FPGA platform specialists. Earlier chapters have presented the positive returns expected to the design project as the team embraces this new methodology. Consider Design-for-Prototyping as the investment made to maximize those returns.

The following sections focus on specific RTL coding style practices and organizational (or procedural) guidelines to achieve this overarching goal.

9.1.1. What's good for FPGA is usually good for SoC

FPGA designs tend to require clean architectures and generic logic for effective synthesis and mapping to the target technology. SoC design processes allow the flexibility of more ad hoc design styles, which may be used inappropriately by some designers to the detriment of subsequent rework.

The need to share a single RTL design for both FPGA and SoC technologies can motivate designers to follow cleaner design styles for simple functionality and reserve unusual coding style for odd cases where it is justified. This path is really the least work for everyone and should benefit the on-going maintenance of the SoC design as a by-product of supporting the FPGA prototype.

The next sections expand on this point with specific coding style suggestions, like FFs at block boundaries, isolation of clock management, and so forth. The whole SoC project should stick to these coding styles, reinforced through formal design reviews and informal organizational behavior. It should be the aim of management and other senior team members who realize the benefits of FPGA-based prototyping to fully embrace the methodology. A “bolt-on” approach to prototyping without a proper integration into the SoC design flow will not yield as great a return on the investment.

9.2. Procedural guidelines

In this section, we outline several important organizational considerations for the prototyping program. There are, of course, many options as to the organizational reporting structure of the prototyping team, however, there should be careful consideration in how the prototyping team’s integration into the organization (and other procedural decisions) affects communication between the design, software, verification, and prototyping teams. Moreover, every effort should be made to integrate and formalize the dataflow and unify the databases, regression, and engineering changes.

Table 22 gives a short summary of the key procedural guidelines that we will cover during this chapter.

Table 22: Procedural recommendations in Design-for-Prototyping

Recommendation	Comment
Integrate RTL team and prototypers.	Have prototyping team be part of RTL design team, trained on same tools and if possible co-located.
Prototypers work with software team.	The software engineers are most common end-users for working prototypes.
Have prototype progress included in verification plan.	Branch RTL for prototype use only at pre-agreed milestones avoiding many incremental RTL dumps.
Keep prototype-compatible simulation environment.	At various points in the prototyping project it helps to compare results with original SoC simulations.
Clear documentation and combined revision control.	Track software and RTL changes for prototyping together using same source control.
Adopt company-wide standard for hardware and add-ons.	Common hardware approach avoids waste and encourages reuse.
Include Design-for-Prototyping in company RTL coding standards.	Most RTL style changes which make prototyping easier are also good for SoC design in general.

The guidelines focus on managing the communication between the different teams during the whole project, including the prototypers as core members of the project from the start of the project (when architectural trade-offs design goals are first being considered).

9.2.1. Integrate RTL team and prototypers

The success of the overall project is going to be improved if everyone can embrace the idea that there is one design with two target technologies. In addition to FPGA skills, the engineers assigned to the prototype will likely be creating supplemental design RTL and making or recommending changes to the original RTL as well. As such, they need to be integrated in the SoC design team or the organization will likely experience duplication of effort and schedule delays. Project management must strive to create a shared responsibility and avoid the behavior of “throwing the design over the fence” at schedule milestones. It is important to have the

prototyping team as part of RTL design team, , trained on same tools, using same RTL coding standards, and having access to same design validation suite (i.e., testbenches and scripts). Involving the SoC design team in the early prototyping effort will also make the prototyping easier.

There are design teams known to the authors in which each SoC designer takes responsibility for the retargeting of their own blocks into FPGA and this is a sure fire way to have those blocks beat the third law of prototyping.

9.2.1.1. Note: FPGA education for SoC RTL team

If management thinks that the design team cannot accommodate an initial prototype implementation, management should strive to cross-train the design engineers to understand both SoC and FPGA technology. By increasing common skill levels it will be easier for engineers to create new logic blocks or introduce engineering changes that will work for both technologies. When technology specific coding is necessary then we should plan it in advance rather than try to redo our design after the event, risking delay in the project.

SoCs and FPGA-based prototypes share many of the same design issues, By developing FPGA skills within the SoC design group we will allow common solutions to these design issues to be established quickly without excessive iterations.

9.2.2. Define list of deliverables for prototyping team

At the start of a prototyping project, the prototyping team needs more than just the RTL. It is important to have a hand-off list of deliverables so that we can check it before the start of the project, or at least understand early on what is missing.. In this way the prototyping team will not waste time later in the project.

Table 23 shows a typical list from a major semiconductor prototyping lab where a design-for-prototyping methodology is being pioneered.

Table 23: Example of design hand-over checklist before prototyping project starts

#	DELIVERABLES FROM DESIGN	REMARKS
1	Updated Device datasheet with <ul style="list-style-type: none"> Detailed system block diagram Detailed Clock generation and distribution block diagram. Design Pin-out with all alternate function mapping. 	Always Required.
2	RTL Database	Always Required. Only RTL files needed. No other views required. It should also include the behavioral models of hard macros. Release should have the RTL source and DOC directories of ASIC database and no other directories.
3	RTL Database Version/TAG and Release Date	Always Required. A TAG of the ASIC database released for FPGA. This will be included in the FPGA versioning register for easy tracking. <i>Mention this in the release note of the database.</i>
4	RTL Database Release notes	Always Required. A text file listing the changes (bug fix/new features) from previous release. This will be released to the users of FPGA platform.
6	HDL File list	Always Required. A list of files read by the simulator. Usually an output of simulator. The same set of files would be read for FPGA synthesis (except for retargeted blocks) thereby ensuring coherency of files between FPGA synthesis and ASIC simulation.
7	ASIC Synthesis Scripts and Timing Constraints in SDC format.	Always Required.
8	ASIC Synthesis Area Report showing module wise area.	Can be delivered at a later stage. Will be used for comparing ASIC gates with FPGA gates and establish a metric for FPGA sizing.
9	Test Suite/Compliance tests	A basic suite of go-no-go tests which can be used to do a quick check of implementation.

9.2.2.1. Reuse file-lists and scripts

It may seem like a simple idea but all scripts and file lists should be kept as part of the source control of the project. By capturing more than just the RTL code it will be possible for others to recreate the build process if needed in the future. The best solution is to create common project make-files for SoC and FPGAs with macro-driven branching for different targets. In this way, the design team captures all key details involved in a “run-able specification.” Ideally, most target specific differences will be isolated to independent modules that can be used as a library for a given target. The alternative of laboring to write lengthy instructions may still fail to document all important setup details and discarding key build scripts is asking for

trouble later. By just establishing simple procedures and “mindset,” this important chore can become routine activity.

The key point that must be addressed by the team is that a single RTL description will be implemented in two target technologies (FPGA and SoC). A conscious effort should be made to separate technology specifics from the intended functional definition. It is important to create clean architectural interfaces with identical behavior in both implementations. Design practices incorporating this idea during engineering change (EC) activity and quickly communicating design impact in a predictable manner to all team members will increase the effectiveness of the project.

9.2.3. Prototypers work with software team

Some organizations building complex systems with new hardware and embedded software struggle with communication problems between the hardware and software groups. Because of the high software content in a modern SoC, it is critical that the FPGA prototype group regards the software group as its customer. Part of the ROI for the prototype is giving software developers early access to functioning hardware and the two groups must have good working relationship to achieve this goal.

A process to track software changes alongside RTL changes will minimize confusion and wasted effort due to compatibility issues. All changes to RTL code should be tracked by a source code control system (like Perforce or SCCS), probably the same one that is used by the SoC and software developers.

Since the prototype will run slower than the final SoC implementation, some areas of the software may need to be rewritten for error-free operation running on the FPGA prototype. Also the designers of the prototype hardware need to consult with the software developers to consider establishing extra probing points, resets or other capabilities that can help in software debug and which are not possible in the SoC implementation. The important point is that early interaction between the two groups can establish extra requirements on both the software design and the prototype design that will help the project achieve its goals and can be included in schedules at the front-end of the planning cycle.

9.3. Integrate the prototype with the verification plan

The decision to use an FPGA prototype needs to be reflected in all aspects of the chip design project plans. This is particularly important in the verification flow. The project should create branches at RTL maturity points for prototype use. Avoid frequent and incremental RTL dumps on prototype designers. The goals for

prototype use in the organization should be clearly stated and reflected in testing plans and project milestones.

Many of the issues concerning planning of the tasks to develop a prototype design are discussed in section 4.2

9.3.1.1. Optimize scheduling of prototype development

The prototype is intended to be a surrogate for the final SoC chip to demonstrate functionality at near-SoC speeds for software developers and others to gain confidence in the correctness of the design. The prototype is not an RTL debug tool – that is the role of the software simulator. We should be careful to wait for the RTL design to reach a pre-agreed level of maturity before starting the prototype development.

There often is a milestone in the design project plan where the team considers the RTL complete enough to issue “hello world” in the simulator. This often coincides with the project milestone when the RTL is ready for a “trial implementation” to identify problem areas in the backend path to silicon. By developing the prototype based on this key RTL milestone will minimize disruption to the prototyping effort from RTL debugging changes and maximize the organization’s use of the prototype.

9.3.1.2. Keep simulation scripts for prototypers to use

As the SoC design and FPGA-based prototype evolve in different directions, we should try to maintain the working simulation models. Then, at key points in prototype bring-up and debug, odd behavior can be checked against working testbenches for the real SoC design. Block-level testbenches in particular, are useful for checking RTL modifications and it is best if the SoC verification team shares key testbenches with the prototyping team.

The goal is to leverage everyone’s experience and avoid needless duplication of effort. Identifying a set of “golden” testbenches will build confidence that everyone is working on the same version of the design.

Assuming that the verification team has been busy while the FPGA-based prototype has been developed, there is a chance that they have already expanded their testbench beyond that running when the RTL was delivered for prototype. It is worth having a procedure in house where simulation results are available for inspection by the prototype team or, better still, the verification team can offer insight into faults seen in the prototype. It could be that the same fault has already been detected and possibly even cured in their RTL already. Regular comparison of prototype and SoC simulation results is recommended.

9.3.2. Documentation well and use revision control

Designers should always strive to write clear, self-documenting code. However in any large design there will be some areas where the intended behavior may be subtle and obvious to the creator but not easily understood by others. In these cases, even a few in-line comments around curious design elements will help. This is especially important for late design fixes to avoid wasted effort due to misunderstandings about assumed code behavior.

The larger organization needs to instill the importance of these efforts across all members of the design team so individuals will take the time to use appropriate coding style with comments when first entering the code. Judgment needs to be used to know when simple code can stand on its own and when naturally complex algorithms or unavoidably tricky code will require extra comments to capture the intended behavior for later maintainers of the code.

On top of this, changes that are made by the prototyping team themselves should be recorded in the same revision control system (RCS) as the rest of the project. We should avoid the prototyping team seeming isolated as “those hackers with the boards” about whom nobody has a clear understanding.

9.3.3. Adopt company-wide standard for hardware

As mentioned in chapters 5 and 6, the choice of hardware platform is crucial to ongoing prototype success. If we are looking to do many prototype projects then it will save a lot of time and money to adopt a standard platform. If boards are in stock or readily available from a standard board supplier then inventory issues need not occur, even when boards become damaged and a quick replacement is required.

If boards and add-ons are compatible across multiple labs and sites then a company-wide knowledge base and expertise can be built up around the chosen platforms. Add-on cards might be developed to attach to the standard base platform for a certain project but would then be reusable across the team and wider company.

The worst-case scenario can be that each team obtains or builds a new and incompatible board for every prototyping project, involving risk and high development costs. Further information about this can be found in Appendix B.

9.3.4. Include Design-for-Prototyping in RTL standards

Most RTL style changes for FPGA-based prototyping are also good practices for SoC design in general (e.g., clear functional architecture, FFs at block boundaries, etc.). Specific guidelines should be incorporated in the company RTL coding standards for future projects. Perhaps just enforcing some existing “motherhood”

standards is now possible with the requirement to map the generic RTL design into two target technologies so there is less temptation to “cut corners” in design specifications.

As experience with prototypes is gained over time designing similar SoCs, it should be possible to find commonality in the prototypes. Consider creating company standards for prototype hardware to encourage reuse and facilitate common add-on devices.

Although the prototype systems are not usually customer-ready products, they should be built to proper engineering standards as they can serve as a reference design for later work and in some cases become the basis of follow-on product designs. In this way the ROI for the original prototype design may be increased several times over.

Even for the original SoC product, if market opportunity creates the need for a slightly modified product, the existence of a reusable prototype system can greatly accelerate development of the new product and reduce risk in verification of the new functionality.

It is a mistake to think of the prototype as a “throw away” step in the process to get to final silicon of the SoC part. Should a major problem be identified in the field, the prototype can be used again to fully verify the engineering changes (EC) that may have caused it.. Following well-documented conventions and practices will enable subsequent design teams to leverage the earlier prototypes when needed.

Examples of coding standards that will benefit the prototype include naming standards for target-specific design elements (such as clock generation, clocks, memories, and analog blocks), check-in regression and linting requirements, and the careful maintenance and enforcement of a concise coding standard document.

9.4. Design guidelines

So far we have explored how procedure can improve the success rate of FPGA-based prototyping with the various teams. Let’s now summarize more technical recommendations, many gathered from other places in this book but some introduced here in this overview for the first time. The following table, which is split across two pages, summarizes the main technical recommendations for the whole SoC project team to follow in order make FPGA-based prototyping a more productive part of the project.

Table 24: Summary of technical recommendations in Design-for-Prototyping

Recommendation	Comment/Detail
Avoid latches	Latch-based designs allow lower power SoC but are hard to time when mapped into FPGA.
Avoid combinatorial loops	Sometimes not seen in SoC RTL because of bottom-up design flow.
Pre-empt RTL changes with `define	`define and `ifdef included in source style guide to include/remove prototyping changes. Use single define for all RTL changes. Used to isolate BIST, memory instantiations, etc.
Low-impact source changes	Always use wrappers and make changes inside those. Replace files, rather than edit them. Back-annotate changes to real source.
Write pure RTL	Allow SoC tool flow to infer clock gating, insert test, apply low-power mitigation etc. avoid instantiating such measures directly into RTL source.
Isolate RTL changes	Make changes inside library elements (RAM, IO library etc.) rather than outside of them in the RTL structure. This improves portability, and places the prototyping code close to the original code it is replacing.
Reuse file-lists/scripts	Common project make-files for SoC and FPGAs with macro-driven branching for different targets.
Memory compatibility	For each new memory generated for SoC, generate FPGA-compatible version. This could be alongside and options controlled with `define.
PHY compatibility	PHY blocks in SoC will need modeling in FPGA, or in off-chip test chip, if available. Keep this in mind when choosing PHY components for the SoC.
Design synchronously	Avoid asynchronous loops, double-edge clocking and other structures which do not map easily to FPGA.
Avoid long combinatorial paths	Use pipelining in RTL to break up very long chains of gates which will run much more slowly in FPGA.
Isolate clock sources in own block	Keep clock gating and switching in own block, preferably at top level. allows easy and complete replacement by FPGA equivalent.

Synchronize resets	Resync reset at each clock domain boundary. Helps avoid race conditions between clocks and reset after partitioning.
Simplify clock networks for FPGA	SoC clock networks are very complex. If possible, implement only a subset of full clocking options for prototyping, use `define`/ifdef to do this.
Synchronize block boundaries	FFs at block inputs and outputs add latency but dramatically simplify timing constraints and increase performance
Synchronize at clock domain crossings	Generally good practice to use synchronizers.
Document any design strangeness	Even a few in-line comments around curious design elements will help. Especially key for late design fix.
Keep simulation environment available	During prototype project, odd behavior can be checked against working testbench. Block-level testbench useful for checking modifications.
Think at architectural level about slow running design	Allows core and peripherals to run at different rates. Allows deep data buffering between slower running FPGA and external data at full speed.
Consider network-on-chip communication vs. wide buses	Industry trend towards “locally synchronous, globally asynchronous” designs will help prototyping.

9.4.1. Follow modular design principles

Modular design is an architectural approach to design wherein special attention is given to creating simple, reusable, and individually distinct functional units to more effectively address the goals of the project. Modular design can and should be utilized from early in the project specification all the way through design and even in the prototyping flow itself. Good modularity in the original RTL structure will affect the prototyping effort.

One example of a modular approach (which has bearing on the prototyping effort) is the specification and design of multiple modular channels. These could be scaled up or down to more or fewer channels, and enable single-core operation of a multi-core (or multi-processor) design. Another example is enabling culling of logic that is not conducive to FPGA adaptation, and considering how this might be accomplished in a way that is simple for the prototype designer without interrupting other project goals.

There are almost always test modes, experimental clock modes, and atypical clock modes that are not required by the prototype. Frequently, it is reasonable to remove them using stub files, or by replacing registers with constants, etc., and then allowing synthesis to cull the coupled logic. The implementation in the flow might utilize ifdef's, libraries, code generators, or constraint files (via RTL modification tools which enable the designer to preserve the purity of the original RTL, but maintain the flexibility to better accommodate the prototyping flow).

The creation of stub files can also be very helpful to the prototyping effort as well as other aspects of the design process when implemented on other functional units. Typically a stub file will consist of whatever interface logic is required to enable the function of the rest of the SoC. In many cases, this may just be a set of constants and pass-through signals to appropriately drive the default values of outputs.

Recommendation: use modular coding style with technology-independent logic elements. Introduce target technology elements only at leaf level or at well-defined functional blocks.

9.4.1.1. Create simple modules

Beyond just using generic logic, it is important to keep the design modules simple and small. Try to avoid making the code excessively general purpose. An RTL design can have excessive complexity when designers attempt to make a module so flexible that it becomes virtually unusable in any specific context. The design of state machines is another example where keeping it simple can be helpful. Some state machine designs span across many pages of case/if statements. This type of design is indecipherable to anyone but the original designer, weak for reuse, prone to error, and difficult to modify. If these machines also include the write-address mechanism for a large register file, it will be very difficult to modify the code in a way that will make the most efficient use of FPGA memory.

Refactoring is another important concept to consider in the SoC design process. Refactoring simplifies code through small redesigns when problems are discovered with the architecture, interfaces, etc. This can include reorganizing functional blocks or dataflow, redefining interfaces or bus protocols, and so on. Refactoring frequently includes extracting common code from one or more blocks and implementing it with more singularity of purpose (often in a single block or function).

9.4.2. Pre-empt RTL changes with ‘define and macros’

The RTL code will need to be modified in some areas for the target technology implementations and this needs to be done in a manner to isolate the change while

preserving the original structure. A good method to pre-empt prototyping changes is to use `define and `ifdef macros. Standard macro names could be adopted and these should be listed in the company-wide source style guide along with guidelines for their usage. In that way a single macro definition at the top-level or in synthesis scripts can be used to allow or mask a large number of RTL changes made to the prototype. Example uses of a macro could be for isolating BIST or memory instantiations.

9.4.3. Avoid latches

While latches can be implemented in the FPGA, they can be inefficient and cause complications in timing analysis. For example, latch-based designs can be used to achieve a lower power SoC but will not be correctly processed by the synthesis FPGA technology mapping tools because the power-saving behavior is not fully modeled in the original RTL.

If latches or FFs could be used to the same effect in the SoC then we should try to use FFs instead of latches; it will simplify the adaptation of the RTL for prototyping. One way to accommodate this type of design is to automate the implementation of latches after the synthesis of a gate-level netlist based on FFs. This may be done on the entire chip, or on an as-needed basis for specialized blocks that require latches. In the later case, it may simplify the task if steps are taken to isolate or otherwise mark the sequential elements requiring conversion.

9.4.4. Avoid long combinatorial paths

If the SoC is designed to be used with the latest technology library then it is very possible that there might be 30 or more levels of logic in some combinatorial paths between sequential elements. This may be perfectly permissible in SoC designs where these levels can be placed very close together and have intrinsically low delay in any case. In an FPGA the logic in these paths will be rationalized as much as possible and mapped into look-up tables (LUTs) but nevertheless, some ten or more LUTs may be needed to create the same critical path. The delay of the LUT is not such a problem as that of the interconnect between them, which might be difficult for the place & route tool to keep short, especially if there are many such paths or the FPGA utilization is too high.

If long paths are expected in the SoC design then it would be preferable if these could be broken into sub-paths by the use of pipelining, which would require some rescheduling in the design.

9.4.5. Avoid combinatorial loops

Combinatorial loops can cause unpredictable behavior in general. Intentional loops to create oscillators or state elements should be replaced with black boxes that can be mapped to technology specific implementations later in the flow.

Unintentional loops can result from incomplete RTL, for example, when not all values are specified in case statement or else statements are missing from if trees or even if default conditions are missing inside always blocks. These would all be caught by even the most superficial simulations, but synthesis tools may generate logic with unintended behavior in these situations.

Loops are sometimes not seen in SoC RTL because of bottom-up design flow and the loops are only completed when the whole design is assembled top-down. Potentially, this first top-down assembly may only occur at the start of a prototyping project. (Note that synchronizing block boundaries eliminates this and many other pitfalls.)

The behavior of circuits containing feedback loops is dependent upon propagation delays through gates and interconnections. Due to process variation and temperature effects, such circuits may be unstable in a given technology and should be avoided.

By carefully re-specifying the logic definition to clearly state the desired function will usually eliminate the combinatorial loop.

9.4.6. Provide facility to override FFs with constants

Consider the possibility of overriding registers with constants wherever possible to increase the adaptability of the SoC design. Examples include configuration registers, such as those used for test modes, experimental clock modes, and atypical clock modes.

These can be implemented with `ifdef macros, code generators, stub files, or constraint files. The synthesis tool will then propagate the constants-eliminating logic that can cause difficulties in the prototyping flows. This can be useful for ATPG flows that are not based on post processing a gate-level netlist and the elimination of clock muxes tied to unused clock configurations, etc. An approach to constant forcing should be taken which leaves the RTL source intact for later use in SoC implementation.

9.5. Guidelines for isolating target specificity

An important modularity concept is the object-oriented notion of “hiding” extraneous levels of definition details from the usage. Using wrappers and other isolation techniques to locally contain or group design elements will help preserve interesting functional reference points in the prototyping flow.

9.5.1. Write pure RTL code

A disciplined effort should be made to create the full RTL description of the chip in terms of generic logic elements or Synopsys DesignWare® components which are well supported by most FPGA vendors. The SoC designer must avoid calling any low-level primitives directly from the target technology library or introduce an explicit clock in the design. Allow the SoC tool flow to infer clock gating, insert test, apply low-power mitigation, etc. Avoid instantiating such measures directly into RTL source.

Designers generally follow top-down design methods owing to successive refinements for implementation, technology-specific details become entangled into the RTL and it is no longer “pure.” For many design teams, that means modifying the RTL description over time so that the actual RTL passed to the prototyping team contains many low-level technology library primitives intermixed with pure RTL elements. This practice will needlessly complicate an FPGA-based prototype and lead to potential errors.

Recommendation: keep the reference RTL design pure and carefully introduce technology specific details only at the leaf level so alternative FPGA and SoC definitions can co-exist in the design database.

By maintaining a pure RTL description of the design it will be possible to isolate all target technology specific details. This is essential to allow sharing of the base RTL code while providing supplemental detail for the FPGA implementation and the SoC implementation separately.

9.5.2. Make source changes as low-impact as possible

Once the generic RTL design has been established, further changes should be done locally without introducing new modularity. Always use wrappers and make changes inside those design elements. Replace files, rather than edit them. Back-annotate changes to real source files.

Make changes inside library elements (RAM, IO library etc.) rather than outside of them in the RTL structure. This improves portability, and places the prototyping code close to the original code it is replacing.

9.5.3. Maintain memory compatibility

Specialized elements or “cores” with optimized implementations are used in FPGAs for bulk memories. It follows that each memory element in the RTL design must be assigned alternative technology specific definitions for FPGA and SoC mapping. This approach will result in the most efficient use of chip resources and insures that some thought will be given to proper modeling of each memory element. For each new memory generated for SoC, just supply a FPGA-compatible version using vendor technology macros. The necessary code for each technology could be side-by-side in the RTL source file and the selection between options controlled with `define.

For the most part, synthesis tools are familiar with the target technology and can map RTL code into FPGA elements in a process known as inference. Whenever possible, having RTL synthesizable behavioral models will improve the FPGA adaptability of our code. There are examples of the use of wrappers and synthesizable memories in chapter 7.

This approach relies on technology mapping algorithms in the synthesis inference tool, however a better implementation may be possible using the library or manual design-file replacement mechanism. One way to do this would be to create a special target direct (perhaps with “_fpga” or a similar prefix) and keep files of the same name as those they are replacing in that directory. With some tools, you can add this list to the end of a current file list, and they will thereby override the original RTL higher in the file list. With others, you may need to create a second file list. There are scripts available from Synopsys that help automate the file list management. This flow should maintain as high a level of target isolation as possible, and avoid having any miscellaneous logic not unique to the SoC/FPGA border.

9.5.4. Isolation of RAM and other macros

It is good practice to introduce an enclosing block or “wrapper” around every technology-dependent element of the design. This includes configured RAMs and other specialized macros in the technology library, which are not generally available in other technologies. Wrappers are allowed break rules about only having logic on lowest levels of RTL as they create a well-defined unit which is replaced with functionally equivalent logic in the SoC design.

The process of including such elements is as follows:

- Create the FPGA element using the tools supplied by the FPGA vendor (such as the Xilinx® CORE Generator™ tool, Memory Interface Generator, etc.). Typically we specify the core type, the target technology, defines the various parameters' initial-states values, etc.
 - The FPGA tool generates the desired core's FPGA netlist and initialization file, where applicable.
 - The netlists are used in the place & route stage, and the template file is used to instantiate the generated core into the main design.
 - In addition, the tool generates a wrapper file containing functional simulation customization data that, combined with the primitive model used in the core, can be used for functional simulation.
 - Add RTL code to instantiate the template file in the design, and connect the module to the design.

Recommendation: use optimized FPGA macros and RAMs to improve FPGA resource utilization and speed, enclosing technology dependent code within a wrapper block to facilitate substitution of SoC implementation and creation of functional test point for verification.

9.5.4.1. Note: handling RAM in formal verification

As we approach the issue of how to implement SoC RAMs in the prototype, we should also address the question of “how do I verify that the behavior of my FPGA RAMs are equivalent to my SoC RAMs?”

Also, the prototype builder should be considering how to verify equivalency through the entire process of converting the SoC design to FPGAs. Doing so will save time later.

The process of using formal verification (FV) on the designs would be much easier if equivalency checking were considered early in the prototyping phase. If we can plan for FV from the beginning with a methodology for verifying the RAMs using testbenches and then plan to formally verify the remainder of the design by black boxing the RAMs, then FV can be a more useful tool. This should be consistent with the use of FV in a general SoC design methodology.

9.5.5. Use only IP that has an FPGA version or test chip

While FPGAs are the main prototyping resource in a typical prototyping system, some SoCs may have a few blocks that either do not map into FPGAs, or blocks for which better prototyping resources are available. Such blocks are typically analog circuits, or fixed digital IP blocks for which neither source code nor FPGA netlist is available. In these cases, we will need to consider solutions outside the FPGA to model the block in the prototype.

IP suppliers typically provide evaluation boards with the IP implemented in fixed silicon. In other cases, the prototyping team may design and build boards that are functionally equivalent to the IP blocks that do not map well into FPGA technology. In still other cases, existing or legacy SoCs may be available on boards as part of the prototyping project and can be added to and augment the FPGA platform.

By using separate hard IP resources for these blocks we will benefit from higher performance (compared to FPGA implementation) and will use less of the FPGA's resources..

Recommendation: use plug-in hard IP “evaluation” devices on the prototype board when available to improve prototype speed and reduce complexity of FPGA logic.

9.5.5.1. Note: PHY compatibility

Embedded IP blocks in the SoC that are provided from the vendor as a physical block without detail RTL description will require special coding in the RTL. If a test chip is available for the IP it should be connected to the prototype board as an external plug-in and the RTL written to use this off-chip connection. Otherwise the IP will need to be modeled with a generic RTL functional definition of the algorithm to be mapped to the FPGA. Keep this in mind when choosing PHY components for the SoC and planning overall architecture of the FPGA prototype. For more information about IP in FPGA-based prototyping see chapter 10.

9.6. Clocking and architectural guidelines

The complexity of today's SoC designs require specific attention be given to overall RTL design architecture related to clocking and managing major synchronous blocks in the system. FPGA technologies are more constrained in the available resources for managing multiple clock domains than custom SoC technology. There will typically be multiple FPGA parts needed to fully model the SoC design, imposing some board-level clocking requirements. Our goal is to abstract some notion of a global clock architecture that can be shared in the RTL description

between both the FPGA and SoC implementations. The following guidelines will help manage this complexity.

9.6.1. Keep clock logic in its own top-level block

If the clock generation logic is kept independent of the rest of the design, it will be more adaptable. Likewise, we should strive to keep unrelated or loosely related design logic out of the clock generation block. Moreover, the independent functional aspects of the clock generation should also be separated into easily recognizable modules. Frequently, there are many more options in the clock-generation logic than are required for the FPGA or even for SoC for that matter. If a block is well separated from other unrelated logic, the block can be replaced by a prototype-specific block that simply drives a few clocks at constant frequencies.

In many cases we may be importing IP with its own internal clocking structure. There may be some value in hard-coding (or at least optionally hard coding) clock selection logic, or otherwise simplifying the clock modules in the IP, if the complexity is no longer required for the specific use model. It may even be possible to completely remove certain components of the clock-generation logic and thereby greatly simplify one of the most involved tasks associated with prototyping.

Keep clock gating and switching in their own blocks, preferably at the top level. This will allow easy and complete replacement by FPGA-equivalent structures.

9.6.2. Simplify clock networks for FPGA

SoC clock networks are often very complex. In general, an SoC has much greater clock flexibility than does an FPGA. If possible, implement only a subset of the full clocking options for prototyping. Use `define`/`ifdef` to control RTL expansion while retaining the full complexity for SoC synthesis.

Simplifying the clock structure is key to the adaptability of the design. Even if your SoC clock structure is thoroughly documented, it may be difficult to implement in the FPGA if it's extremely complicated. One of the biggest contributors to clocking complexity is test logic. Often automatic test pattern generation (ATPG) circuitry inserts logic to multiplex clocks. Note that an FPGA, being reprogrammable, ships fully tested. There is almost never a reason to include ATPG logic in the FPGA prototype. Including it introduces unnecessary complexity to the FPGA clocking structure. If the insertion is automated it can easily be disabled for the FPGA implementation.

However, some test logic may be required in the FPGA prototype. For example, almost all microprocessor designs include some sort of serial debug interface that

works through a JTAG clock. This is absolutely necessary for doing software development on the FPGA. It is helpful to partition the test logic into that which is necessary for only the SoC and that which is necessary for the FPGA. At a minimum include comments in the code to indicate which is which. Also indicate in the comments how signals should be tied off to disable test logic that's unnecessary for the FPGA. Ideally, insert “`ifdef FPGA ... `else ... `endif`” pre-processor commands to separate functionality required for the FPGA prototype from that required for the SoC. In the “`ifdef FPGA`” clause, tie off unnecessary test logic inputs to their disabled values so that this logic is pruned in FPGA synthesis.

9.6.3. Design synchronously

Avoid asynchronous loops, double-edge clocking and other structures that do not map easily to FPGA. Limit the overall design to conventional synchronous design methods. If unusual structures are required in the SoC design then isolate those circuits to local blocks, which can be replaced with equivalent functions in the FPGA design.

Maximizing the use of conventional synchronous design style will greatly simplify the effort required to develop the FPGA prototype.

Minimize the portions of the design running on clocks required for external interfaces, and use asynchronous FIFOs whenever possible to transfer data to and from these interfaces to the system clock domain.

9.6.4. Synchronize resets

Remember that the FPGA configuration process initializes every block RAM, distributed RAM, SRL, and FF to a defined state even if no reset is specified in RTL, so explicit code is not required as it is in the SoC. Depending on how the resets and presets are defined, they can have a significant impact on what can be inferred and therefore how much of the FPGA’s special resources can be automatically used. The key point is to spend more time earlier considering the reset strategy, and write the reset logic in a way that is simple, consistent, and flexible. If some of these FPGA considerations can be accommodated it will enable many creative solutions.

9.6.5. Synchronize block boundaries

Use FFs at all block inputs and outputs. This practice will add latency and require rescheduling of modified paths with respect to the rest of the design, but it dramatically helps to apply timing constraints and to meet timing targets. It also

assists in chip layout and FPGA partitioning. Certainly we should already be synchronizing at clock domain crossings.

Often it is not practical to modify the design scheduling specifically for the FPGA. However, if the common design practice of inserting FFs at the boundaries of each designer's block is observed, the likelihood of having FFs between partitions is substantially higher. If we follow the practice of synchronizing boundaries then we should have far fewer exceptions to handle when it comes to constraining the FPGA prototype.

9.6.6. Think how the design might run if clocked slowly

Prototypes are constructed from multiple FPGAs, a printed circuit board, custom IP core plug-ins, and other components. The design architecture of this mixed-technology solution must allow the cores and peripherals to run at different rates. Thought must be given to the interfaces between all of the elements and whether we need to provide deep data buffering between slower running FPGAs and external data that is running at full speed.

If addressed at the architectural level, the expected speed differences between SoC and prototype implementations can be cleanly isolated and managed properly. For example, in many cases rate adapters are required for the FPGA prototype. Often, if considered early in the architectural phases of the design, major functional modules' bus interfaces can be designed such that speed bridges are not required. Because of the independence afforded by this architectural style, resultant designs tend to be more robust, adaptable, and readable as well.

9.6.7. Enable bottom-up design flows

The ability to easily implement a bottom-up design flow can be very advantageous to the implementation of the prototype. Many of the design and architectural recommendations already mentioned will naturally enable bottom-up design flows. (The concepts of synchronicity, simplicity, and isolation all typically provide benefits to these flows.) We can enhance bottom-up flows further by considering how synthesis tools create automated enhancements.

If high-level modules are kept free of parameters and generics, a bottom-up flow will require less effort from the tools or engineers attempting to pre-process and unify the modules.

An exception to the preference of keeping clock logic in its own top-level block may also be considered to more easily enable clock conversion in bottom-up flows. If gated-clock conversion is being implemented, we may want to consider moving the final on/off gating to the module which is being selectively disabled, so that this type of conversion can be automatically handled by the synthesis tool without error-prone manual hierarchical modifications.

9.7. Summary

The main point of our Design-for-Prototyping manifesto is that the use of FPGA prototype reshapes the development task by providing a confidence-building “executable specification” which, through its speed and RTL clarity, empowers the individual groups within the design team to work more effectively to achieve the SoC design project goal.

The key proposals of Design-for-Prototyping are:

- Development of FPGA prototype is a key element in the overall SoC design project and so needs to be included in plans and schedules.
- The RTL design needs to follow a robust coding style to effectively represent both FPGA and SoC technologies, both in first coding and on-going refinements. The resulting quality of RTL definition will pay dividends throughout the life of the design.
- Use modular coding styles including clean separation of prototype-specific components from the rest of the design, independent dataflow, and isolation of clock domains.
- Expand design documentation to identify as early as possible challenging parts of the design to the prototyping team.
- The SoC Team might need to realign slightly in order to integrate prototyping in their processes and staff skill sets to maximize the possible benefit.

The end result of these seemingly obvious but perhaps arduous changes will be that FPGA-based prototyping benefits will be derived earlier in the project, enabling earlier software validation and pre-silicon integration.

The authors gratefully acknowledge significant contribution to this chapter from
Mark Nadon of Synopsys, Austin

So far we have explored the tasks involved in porting an SoC design onto an FPGA equivalent form and how following some Design-for-Prototyping rules, these tasks can be made easier. A significant part of the porting effort will be put towards handling pre-existing intellectual property (IP), and particularly peripheral IP for interfaces in the design and so it is deserving of its own chapter. In this chapter we shall explore the two most popular groups of IP, namely CPU and interfaces and how we can model these in an FPGA-based prototype given that we may or may not have access to hard models, to RTL or even to an equivalent IP already available in FPGA-compatible form.

10.1. IP and prototyping

Almost all SoC designs today include some form of IP, which we shall define as design blocks not originating with the project but instead brought in from “outside.” We are limiting this discussion to digital IP because analog or mixed-signal IP will clearly not function in an FPGA and will need external support (see chapter 4).

From our prototyping perspective, it is valuable to know that the IP components are pre-tested and should work as specified. If so, then our prototyping task becomes somewhat easier, however, the prototype is an excellent platform for testing the combination of the IP blocks and their interconnection. Our SoC design is probably the first platform in which a particular combination of IP blocks has ever been used together. As we shall see below, enabling our prototype to properly model the combination of IP blocks may require different approaches depending upon the format in which the IP is supplied. Some will be modeled inside the FPGA while others may require external test chips. In each case, the IP supplier should be willing and able to support its modeling in an FPGA-based prototyping environment. It is of great value in SoC design to know that the IP block is tried and tested on silicon but this should also be true for the prototyping options, such as a test chip or a working FPGA image. The SoC team should ask its IP supplier what help they can offer to the prototyping team.

Our challenge in linking from IP in our FPGA-based prototype to other forms of IP outside of the device is to remain functionally equivalent to the original SoC design.

10.2. IP in many forms

Digital IP may take many forms and originate from many sources, internal and external. The obvious examples would be a CPU core from third-party suppliers such as ARM®, or peripheral IP from Synopsys. Smaller elements such as the DesignWare® Building Blocks from Synopsys or generated by the Xilinx® CORE Generator™ tool are also examples of IP and have very widespread use in SoC and FPGA respectively. Even the reuse of another designer's block can be considered IP as long as it is packaged with all the documents, infrastructure and support necessary to bring the IP through the whole SoC tool flow. However, that is where we draw the line for the purposes of this discussion – just using somebody else's RTL in our design does not qualify as IP.

IP is delivered in a number of forms, each of which presents us with a different challenge. The main formats are listed in Table 25 and most methods of IP delivery will fall into one of the listed categories.

Table 25: Formats for IP delivery to SoC Team

RTL source code	Whole source code in Verilog HDL or VHDL, either under vendor license or from open-source provider.
Encrypted source code	RTL is protected and must be decrypted either explicitly or as part of the tool flow. Once decrypted, it behaves as any other RTL source.
Soft IP	Delivered in an intermediate form, sometimes encrypted, but requiring back-end processing.
Netlist	IP delivered as a pre-synthesized netlist of either SoC library elements or generic gate-level elements, such as Synopsys GTECH.
Physical IP	Also known as hard IP. Pre laid-out by a silicon foundry. Would be represented by models or test chips during development and prototyping.

Any mix of these types of IP may be found in any given SoC design. Sooner or later we may need to address all of them in our prototyping projects. Let's look closer at each of these forms of IP in turn and explain how each can be handled in an FPGA-based prototype.

10.2.1. IP as RTL source code

When we have the RTL source code for an IP block, our task would seem to be no different than it would be for any other RTL in the SoC design. There are, however, some differences in how much we can understand or alter the RTL within an IP block. For example, it may be that the RTL is delivered under a license that governs that it may not be altered without voiding any warranty or support agreements. In that case, the IP vendor could be asked for a separate license or consulting contract for support of prototyping.

Alternatively, the IP vendor may already have an FPGA-ready form of the IP but that may require an extra license agreement, possibly at extra cost. When choosing IP for the original SoC design, some thought might be given to the availability of FPGA versions of the IP from the vendor.

Licensing and support agreements aside, as long as the RTL is complete and well documented, there is no fundamental reason why we might not successfully prototype it along with the rest of our SoC design. Although the functionality of the IP will be the same in the FPGA, as with other SoC-targeted RTL, we should lower our expectations regarding performance. RTL that is developed and tailored for a leading edge SoC process will run considerably slower in even the fastest FPGA today.

10.2.2. What if the RTL is not available?

One of the issues with supplying IP as RTL to large numbers of people within a company or to any third-party developers is IP pollution and even theft. IP pollution is the accidental or even deliberate alteration of the IP for a design-specific purpose without the knowledge of the IP vendor, leading to licensing and support issues. IP theft needs no explanation and is a key issue for all IP vendors and RTL supply is something that is very carefully monitored and protected between supplier and customer.

As RTL is so valuable, many IP providers and users minimize their exposure by using a secure method of delivery, limiting the spread of the RTL and the possibility of “reverse engineering.” Methods used for this include supplying the IP in more abstracted formats such as simulation models, encrypted netlists, encrypted FPGA bitstreams and full-silicon test chips. The advantages and limitations of each are listed in Table 26.

Table 26: Advantages and limitations of IP delivery formats in place of RTL

IP format	Advantage	Limitation	Security
Model	Great for simulation.	Performance limited by simulator. Cannot be used for prototype	“Secure” since it is not the real IP.
Encrypted FPGA netlist	Harder to reverse engineer than raw RTL.	No visibility for simulation	Security only as good as encryption.
Encrypted FPGA bitstream	Faster than simulation.	Less flexible and may involve compromise	Very secure.
Test chip	Highest performance.	Lowest flexibility. Long development.	Very secure.

A model is really only a representation of the IP for the purposes of simulation. Whether written in RTL or in a higher-level format such as SystemC, it is not intended for synthesis or implementation in FPGA or any other physical form. Models are nevertheless required, not only for simulation but also because they recover some of the visibility otherwise lost by using an encrypted or test chip form of the IP. For example, a test chip of some IP may be delivered with a transaction-level model (TLM) for inclusion in a high-level testbench but we should not synthesize the TLM into silicon. A good measure of the maturity of an IP block (and indeed a good measure of the IP vendor) is the amount of extra support, such as models, provided with the IP.

In the case of CPU IP, models may also be provided for instruction set simulators which have very limited knowledge of the cycle accuracy of the IP but are very fast and ideal for use in software development before RTL is available. As we shall explore in chapter 13, models running on functional, virtual or instruction-set simulators can be interfaced to real hardware using standards like SCE-MI to give a solution partitioned between models and the real SoC RTL.

Let us now consider how we can prototype with these forms of IP for which we do not receive the raw RTL.

10.2.3. IP as encrypted source code

Beyond the relatively trivial example of including IP as RTL source code we find the first degree of difficulty is in RTL which is delivered only in an encrypted form. This means that we need the license to use the IP and the decryption mechanism to

access it. IP vendors will each have their own approach to encryption and decryption, including common public domain methods such as PGP (“pretty good privacy”) as well as proprietary methods. If the protection is only used for shipping the source code then after decryption we will be in possession of the RTL, and, as above, fully able to proceed with our prototyping work.

However, some RTL is not only shipped encrypted but remains encrypted throughout the tools flow, automatically decrypted “on-the-fly” at each step as required. This is only possible if each tool has built-in understanding of the required decryption and the necessary keys. Two common examples of such an approach are the synenc encryption flow from Synopsys® and the encrypted nge files generated by Xilinx® tools.

The overall aim is to get the source code for the IP of possible, or create some route where the IP instantiation black box can be filled. This is an exercise that we will not go into in any further here except to say that the industry will eventually move towards an IEEE standard for IP encryption and encapsulation.

10.2.4. Encrypted FPGA netlists

Encrypted netlists can be supplied to companies for inclusion into their own FPGA designs but offer a low level of security since they are inevitably decrypted in the design flow. The resulting output from the design flow is encrypted again to ensure the final image file can only be used in a deterministic way in the target FPGA hardware.

FPGA vendors offering encrypted IP (from third parties) for their products rely on different techniques to limit the use in the target FPGA hardware, this can range from IP that will only work for a limited time when programmed into an FPGA and/or requires the use of the download/debug hardware and software used to configure the FPGA. These are special netlist designs with additional logic to perform checking and security. They still rely on a legal agreement to ensure the final level of protection of the IP.

10.2.5. Encrypted FPGA bitstreams

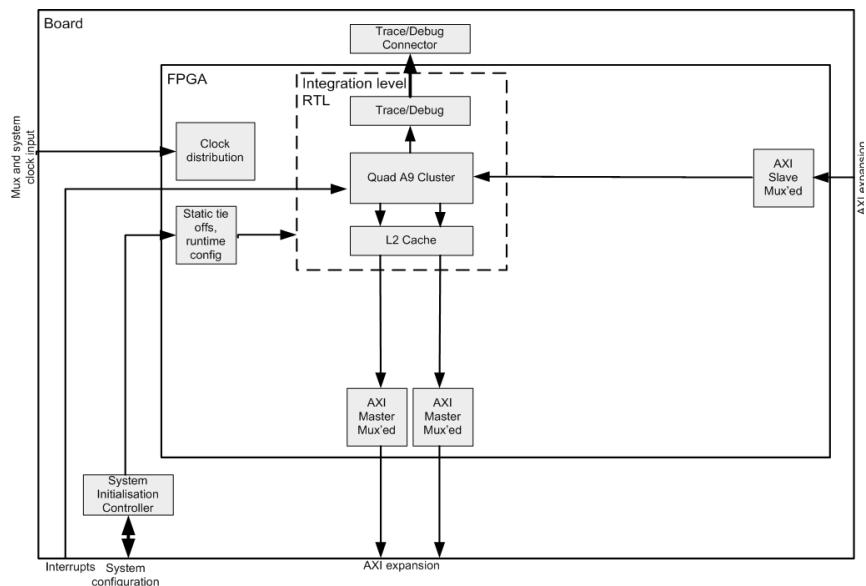
Some FPGAs have built-in encryption keys which are used to decrypt an encrypted FPGA bitstream on the fly as it is read in by the FPGA and configured. The latest -6 devices and the largest Xilinx® Spartan®-6 devices all use AES 256-bit encryption/decryption. The decryption key is stored in battery-backed key memory, or it may also be stored in a less secure poly eFUSE key. The battery-backed key is the most secure, as there are no known means to obtain the key from a device. Powering the key memory off causes the key to be lost completely. The eFUSE key

is less secure, as destructive tear-down of the device may be used to read the value of the eFUSE bits. However, even then the eFUSE bits are not easy to read since there are three times as many bits used than are strictly required for the security key, further confusing the would-be key-copier.

This encryption methodology was originally intended to stop design cloning of FPGA-based products, but it is also a useful way to secure IP blocks supplied in FPGAs. Since all the decryption is performed within the FPGA and only the encrypted FPGA image is visible outside then the security of this is very good and an excellent method for vendors to deliver high value IP.

Figure 123 (courtesy of ARM) shows how the FPGA image would replicate the integration (top) level of an IP block design. This would effectively be the same

Figure 123: Top-level of IP used as encrypted FPGA image (source: ARM Ltd.)



Copyright © 2011 ARM Ltd.

interfaces that would be exposed when using a hardened macro from the silicon provider or when we have hardened an IP block to our requirements. Due to the high number of signals that are normally associated with this level of the design, there may be a requirement to multiplex these signals to and from the FPGA. This then requires the opposite logic to reconstruct the signals to join them to the rest of the design. The vendor supplying the FPGA image should also provide the application notes and support to enable us to do that.

A useful feature of IP delivered in this form is that the clocks in the FPGA do not make use of the internal MMCM and such elements. This allows the system to be clocked at speeds below that minimum limit that would have been imposed had they been present. Indeed, this approach may even support clock stopping and single stepping in the IP block.

As users then, we would need to provide the clocks for the different domains in the FPGA's internal logic and the pin multiplexing and so forth for our prototyping system in order to ensure proper clock alignment.

This approach is used by ARM in its software macro models, or SMMs, which are encrypted FPGA images. ARM feels that the omission of clock infrastructure gives the SMM a greater operational flexibility, which supports a more end-user applications without the need for altering, or even viewing, the RTL.

10.2.6. Test chips

Probably the most secure method of delivery for the IP vendor is for the design to be pre-implemented in silicon, as it is very hard to reverse engineer. However, it is also the most costly to create, maintain and support, especially if the vendor has to build a new test chip for each revision of its IP. Test chips are usually available for higher value IP blocks with wide usage, for example, most ARM CPU cores have test chips. It is less likely that a test chip would be available for either a new block (e.g., supporting a very new communications standard) or for a specialized IP block which is customized by the vendor for each user.

Test chips may require associated components to support their operation (e.g., memory controllers) but the combination of external test chip and support will allow the FPGA-based prototype as a whole to run at the highest speed possible. However, to achieve these higher speeds we often need to compromise on features

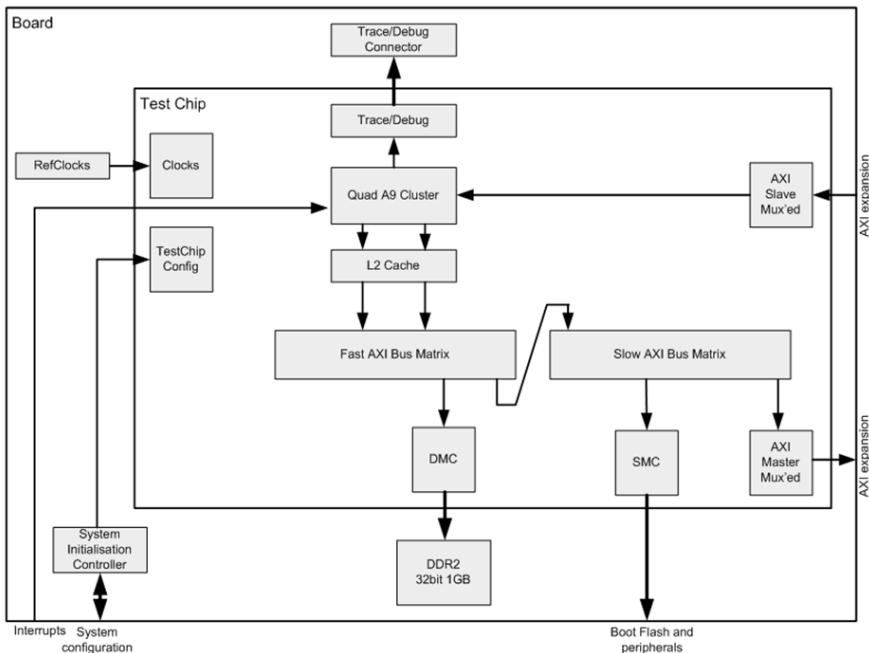
The biggest limitation of the test chip is its lack of flexibility, owing to pre-defined interfaces and configuration options. This may impose restrictions on usage, for example memory maps or interrupt structures which may not match the expected use of the IP in the final SoC. Compromises to IP in order to allow its use in test chips include selective adherence to cycle accuracy, use of asynchronous bridges between the test chip and the FPGA, multiplexing of signals to accommodate the high pin count buses and even limiting the features of the implementation to meet the prototyper's needs or silicon limitations (e.g., disabled test modes, less interrupts, merging buses on chip to bring a single bus to the pins, etc.).

Software or system settings would need to be altered to match the test chip's capability, rather than the other way around and it may be that a more flexible RTL or FPGA-based delivery is required. This might mean that we need to obtain extra licenses from the IP vendor compared to a test-chip and model approach, but it may

be worth the investment if it means that the FPGA-based prototype is going to be more useful with it.

Figure 124 (courtesy of ARM) gives an example test chip implemented for the same ARM processor example shown earlier in Figure 123. Here we can see the use of the SMC (static memory controller) and DMC (dynamic memory controller) to access to boot memory and peripherals, together with the DMC for run time memory.

Figure 124: Top-level of test chip equivalent to Figure 123 (source: ARM Ltd.)



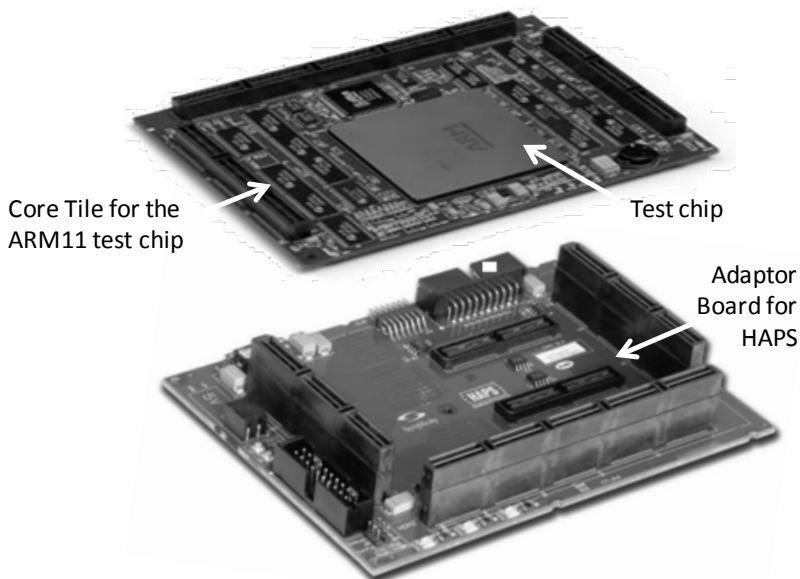
Copyright © 2011 ARM Ltd.

Test chips are able to support benchmarking and OS development in the early stages of a design ensuring that we can make an early start on the software. However having all these features does limit the flexibility of the test chip in the hardware prototyping system (due to memory map, interrupt and fixed configurations). The test chip of the kind illustrated is generally best placed to support OS development, benchmarking activities and development of extension IP blocks which will be connected via the AXITM bus (in the case of the ARM).

10.2.7. Extra FPGA pins needed to link to test chips

As we saw in chapter 5, we recommend avoiding permanently linking FPGA pins to peripheral or other external components on the board. Instead we should keep such connections flexible in order to increase the chances for their reuse in future prototyping projects. In the case of IP test chips, we may need to connect our FPGA(s) to a great number of external pins on the test chip or a board/module upon which the test chip is mounted. We should try to make such connections via deferred or switched interconnect (see chapter 6) and this may involve adaptors or vendor-specific connectors.

Figure 125: ARM test chip on a CoreTile with associated adaptor for Synopsys HAPS®



An example of an ARM test chip mounted on a CoreTile and its associated adaptor with which it would communicate with a Synopsys HAPS® FPGA board is shown in Figure 125.

The use of wide buses in SoCs normally means that the connections between FPGA and test chip need to be multiplexed in order to reduce the number of FPGA pins required or to simply fit within the number of pins of the connector. This however, will also introduce extra delay of the interconnect IO pads and boards, probably reducing the overall system speed. More discussion of multiplexing and its impact on timing can be found in chapter 8. This can be mitigated to some degree by using high-speed serial signaling techniques and higher speed multiplexing rates (see HSTDIM discussion in chapter 8).

Having discussed the different formats in which IP can be delivered, we shall explore the handling of soft IP and hard IP in more detail and also explain how these can be included in our FPGA-based prototype.

10.3. Soft IP

Soft IP can be any form of IP for which physical implementation is decided upon by the end-user. For example, IP delivered as RTL can be considered “soft” because we are at complete liberty to compile, synthesize and lay-out the IP in any way that

Figure 126: Example of soft IP: datasheet of a MAC from a DesignWare library



DW02_mac
Multiplier-Accumulator
Last Revised: Release DWF_0703

Features and Benefits

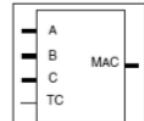
- ❖ Parameterized word length
- ❖ Unsigned and signed (two's-complement) data operation
- ❖ Inferable using a function call

Description

DW02_mac is a multiplier-accumulator. It multiplies a number A by a number B, and adds the result to a number C to produce a result MAC.

The input control signal TC determines whether the inputs and outputs are interpreted as unsigned ($TC=0$) or signed ($TC=1$) numbers.

To extend the accuracy of the accumulator beyond $A \times B$, use DW02_prod_sum1 Multiplier-Adder in place of DW02_mac.



Pin Name	Width	Direction	Function
A	A_width bit(s)	Input	Multiplier
B	B_width bit(s)	Input	Multiplicand
C	$A_width + B_width$ bit(s)	Input	Addend
TC	1 bit	Input	Two's complement control 0 = unsigned 1 = signed
MAC	$A_width + B_width$ bit(s)	Output	MAC result ($A \times B + C$)

we choose. Therefore, in our earlier discussion of the various RTL delivery, with or without encryption, we were in fact exploring soft IP.

However, any form of IP for which we do not receive layout or any other physical information should be considered “soft.” For example, netlists or binary forms of the IP, or IP that is pre-compiled into an intermediate library. Relatively low value IP or IP for which performance and area targets are reasonably easy to achieve are often delivered as soft IP.

A very common example soft IP is the DesignWare Building Block library from Synopsys. DesignWare is an extensive library of infrastructure IP for design and verification including arithmetic and datapath components, AMBA interconnect IP and microcontrollers. The datasheet for an example DesignWare component is shown in Figure 126 where we see an excerpt from the datasheet of a multiply-accumulate function, or MAC.

SoC designers will make use of such a soft IP block in one of three ways: instantiation, inference or operator replacement. Instantiation is the simplest, with the block’s module described and instantiated in the normal way. The descriptions would be available from a library or directly in the RTL. An example instantiation only is shown in the code excerpt in Figure 127.

Figure 127: Instantiation of DesignWare® MAC IP in Verilog HDL

```
module DW02_mac_inst( inst_A, inst_B, inst_C, inst_TC, MAC_inst );

parameter A_width = 8;
parameter B_width = 8;

input [A_width-1 : 0] inst_A;
input [B_width-1 : 0] inst_B;
input [A_width+B_width-1 : 0] inst_C;
input inst_TC;
output [A_width+B_width-1 : 0] MAC_inst;

// Instance of DW02_mac
DW02_mac #(A_width, B_width)
  U1 ( .A(inst_A), .B(inst_B), .C(inst_C), .TC(inst_TC),
    .MAC(MAC_inst) );

endmodule
```

DesignWare IP may also be inferred as function calls. This relies upon the inclusion of support references in the RTL (via and `include statement in Verilog or library reference in VHDL). In the example shown in Figure 128, the tool must also be set up so that a search path points to the files to be included. Here we see that one of two different functions are included each inferring either a two’s complement or an

unsigned version of the MAC. The synthesis combines these together into a common MAC with configurable least-significant-bit. Different forms of the IP may have been created, for example a power-optimized or a performance-optimized version of a multiplier. The user or the tools would have the ability to choose the most appropriate version in the context of the design at compile time.

Some tools may even infer soft macros synthetically during operator replacement or during a sub-function of synthesis called module compilation. For example, a multiplier soft IP macro would be inferred by the code simple code $c \ll= a * b$ with the result that the gate-level netlist would have an extra level of hierarchy containing gates optimized for the target constraint.

Considering the value of soft IP, it is not surprising that most SoC designs today include such elements and so we need to be able handle them during FPGA-based prototyping. Continuing with DesignWare as an example of soft IP, each of the above three methods for including the IP in the SoC design will require different solutions for their correct operation in FPGA.

Figure 128: Inference of DesignWare® MAC by function call

```
module DW02_mac_func (func_A, func_B, func_C, func_TC, MAC_func);
parameter func_A_width = 8;
parameter func_B_width = 8;

// Pass the widths to the multiplier-accumulator function
parameter A_width = func_A_width;
parameter B_width = func_B_width;

`include "DW02_mac_function.inc"
/* requires search path to point to file */

input [func_A_width-1 : 0] func_A;
input [func_B_width-1 : 0] func_B;
input [func_A_width+func_B_width-1 : 0] func_C;
input func_TC;
output [func_A_width+func_B_width-1 : 0] MAC_func;

assign MAC_func = (func_TC) ? DWF_mac_tc(func_A, func_B, func_C) :
                           DWF_mac_uns(func_A, func_B, func_C);
endmodule
```

10.3.1. Replacing instantiated soft IP

An IP instantiation may not be understood by the FPGA synthesis in the same way as the SoC synthesis, if at all. In most cases, the IP instantiation would appear as a black box, requiring contents at some point in the FPGA flow. If we are able to advise the SoC designers during their initial choice of IP, then we should ask them to ensure that only IP with an available and proven FPGA equivalent is chosen. In this way the original SoC design could have `ifdef branching between two instantiations, based on a single `define variable.

Perhaps even more preferable would be a single instantiation which serves both SoC and FPGA purposes, with the tools in the two different flows providing the appropriate contents for the instantiation. In that case the SoC designers would not be required to make any special provision in their RTL, except to choose only from a supported library of soft IP for which FPGA equivalents are available.

One way to provide this in the FPGA flow is for the prototyping team to write an additional RTL file with the same functionality as the soft IP. To enable this more easily, we would use wrappers in the original SoC design, as we did for memories in chapter 7. We could ask the SoC team that each time they instantiate a soft IP element that it is placed in a wrapper so that it can easily be replaced with the FPGA equivalent. This is another example of good Design-for-Prototyping practice as discussed in chapter 9.

A more automated approach may require some investment in time and effort, but if a particular soft IP library is to be used often, then it would be worth the investment. For example, at Synplicity®, Bangalore, before the acquisition by Synopsys, each DesignWare building block was analyzed and functionally equivalent RTL was created for use in the Certify® tool. In that case, all properly instantiated DesignWare elements would be automatically interpreted, not as a black box but as a new bottom-level of the RTL hierarchy. This was not particularly optimized for FPGA, but during FPGA synthesis was interpreted in the same way as any other piece of RTL.

More recently, Synopsys has modified its FPGA synthesis tools to allow native use of the DesignWare blocks for FPGA designers. In addition the DesignWare IP developers themselves have also performed some optimization for the blocks to better operate in FPGA. Any instantiation or inference of a DesignWare building block element in an SoC design will also be automatically interpreted correctly by the FPGA synthesis tools.

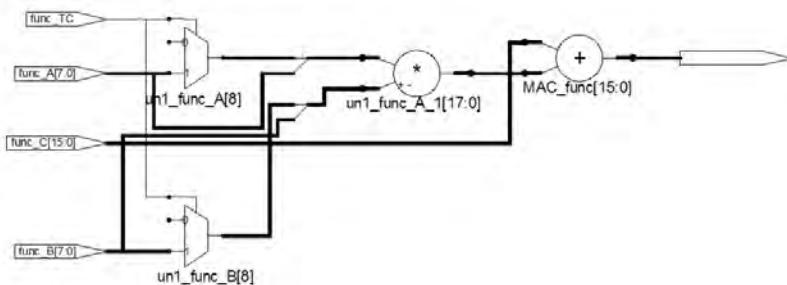
10.3.2. Replacing inferred soft IP

The result for inferred soft IP is very similar to that for instantiated, however, now it is not a matter of “filling” an empty black box but of the FPGA tools inferring the

same functionality as the original SoC synthesis for a given piece of RTL. In both cases, the library references and/or `include statements must be resolved to the respective target's implementation. In the example RTL in Figure 128, the SoC synthesis tool (Design Compiler[®]) has its search path set up to include the path to the functions for all the DesignWare used in the design. When this same RTL is passed to the FPGA tool during the prototyping project, the equivalent path variable needs to be set to point to a set of functions for the DesignWare used. Alternatively, if the DesignWare elements are few in number and rarely used, then the required extra function definitions could be placed in a local include file.

The function definitions included will not be identical to the design blocks used to fill the instantiations mentioned in section 10.3.1, however, we refer to other sources for how to use functions in Verilog HDL.

Figure 129: Logic automatically included for DW02_MAC functional inference



Referring back to our dw02_mac example in Figure 128, if we synthesize that in FPGA synthesis then the resultant logic created is shown in Figure 129 and it is simple to see how this would be mapped into FPGA.

10.3.3. Replacing synthetic soft IP

In SoC synthesis, an RTL operator – whether built into the language, like +, -, and *; or user-defined, like functions and procedures – can be linked to a synthetic operator. A synthetic operator is an abstraction that makes it possible for the synthesis tools to perform arithmetic and resource-sharing optimizations before binding the operation to a particular synthetic module.

The linking mechanism will vary from tool to tool but in Design Compiler it is a recognized HDL comment called a pragma. When the compiler sees the pragma, the *map_to_operator* as a comment in the RTL, then the logic is used in place of the operator. Operator inference occurs when the synthesis tool encounters an HDL operator whose definition contains a *map_to_operator* pragma. The tool finds the

specified synthetic operator, inserts it into the user's design, and performs high-level optimizations on the resulting netlist. In fact, this is the mechanism used for the functions in the example of inferring the dw02_mac in Figure 128.

Table 27, taken from the DesignWare Developers Guide, lists the HDL operators that are mapped to synthetic operators in the Synopsys standard synthetic library for Design Compiler (for more information in the references).

When soft IP is inferred by the SoC synthesis tools in the above way from generic

Table 27: Synthetic soft IP mapped to operators

HDL Operator	Synthetic Operator(s)
+	ADD_UNS_OP, ADD_UNS_CI_OP, ADD_TC_OP, ADD_TC_CI_OP
-	SUB_UNS_OP, SUB_UNS_CI_OP, SUB_TC_OP, SUB_TC_CI_OP
*	MULT_UNS_OP, MULT_TC_OP
<	LT_UNS_OP, LT_TC_OP
>	GT_UNS_OP, GT_TC_OP
<=	LEQ_UNS_OP, LEQ_TC_OP
>=	GEQ_UNS_OP, GEQ_TC_OP
if, case	SELECT_OP
division (/)	DIV_UNS_OP, MOD_UNS_OP, REM_UNS_OP, DIVREM_UNS_OP, DIVMOD_UNS_OP DIV_TC_OP, MOD_TC_OP, REM_TC_OP, DIVREM_TC_OP, DIVMOD_TC_OP
=, not=	EQ_UNS_OP, NE_UNS_OP, EQ_TC_OP, NE_TC_OP
<<, >> (logic)	ASH_UNS_UNS_OP, ASH_UNS_TC_OP, ASH_TC_UNS_OP, ASH_TC_TC_OP
<<<, >>> (arith)	ASHR_UNS_UNS_OP, ASHR_UNS_TC_OP, ASHR_TC_UNS_OP, ASHR_TC_TC_OP
Barrel Shift ror, rol	BSH_UNS_OP, BSH_TC_OP, BSHL_TC_OP BSHR_UNS_OP, BSHR_TC_OP
Shift and Add srl, sll, sra, sla	SLA_UNS_OP, SLA_TC_OP SRA_UNS_OP, SRA_TC_OP

RTL then its replacement for prototyping is an almost trivial task. The same RTL which infers the soft IP in the SoC synthesis will be automatically interpreted by the FPGA synthesis tool and mapped into relevant FPGA resources. For example, the SoC synthesis might employ a synthetic operator bound to a dw02_mult block in order to represent the * in a simple statement $c \leq a * b$; . The exact same * will be inferred as a multiplier by the FPGA synthesis and mapped to a dedicated FPGA multiplier resource by default.

Because of this automation and simplicity, SoC teams should try to employ synthetic operators as often as possible rather than instantiate the soft IP directly into the RTL.

10.3.4. Other FPGA replacements for SoC soft IP

Xilinx has a large variety of IP which are licensed for use only within their own FPGAs. We can look at the functionality of the soft IP in the SoC design and find a close, or maybe even exact, equivalent in the FPGA library. Of course its use would be only temporary for the sake of prototyping but it may be that we can use a wrapper in the same way that we do for RAM. The more complex the IP, the more useful that this would be, but also the less likely that a match can be found between the SoC IP and the FPGA IP. The one exception to this is in the area of standards-based peripheral IP. Let's look at that more closely now.

10.4. Peripheral IP

So far we have discussed handling of small elements or blocks of IP embedded in our RTL. For relatively low-level elements such as the DesignWare Building Blocks, the creation of an FPGA equivalent is a reasonable task but what of the sub-system level IP or whole CPU cores or peripheral functions, such as PCIe or USB? In some cases, we may need to spend a substantial effort replacing this kind of core IP with an FPGA equivalent or external component so before we do, we should be sure of our purpose.

There are a number of reasons high-speed IO needs to be prototyped and we summarize these in Table 28.

Table 28: Reasons and considerations in using peripheral IP in a prototype.

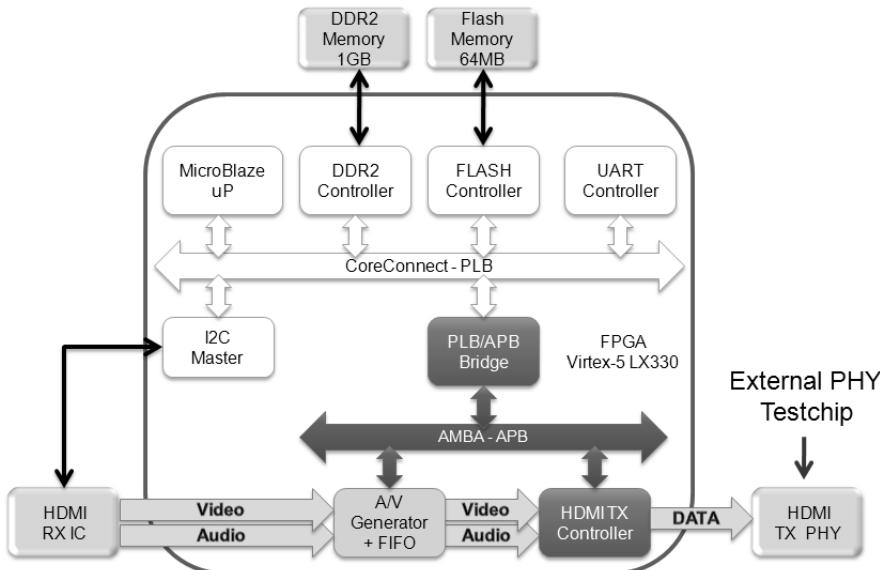
Use Model	Speed	Accuracy
Prototyping IP itself	Highest speed	Cycle accurate
Prototyping IP as part of larger SoC	Scaled speed	Cycle accurate
Software validation	“Enough speed”	Functional equivalence

10.4.1. Use mode 1: prototype the IP itself

The prototype project is mainly performed in order to verify the IP design itself. In that case, our task is actually a small-scale version of a normal SoC prototyping project. We would prefer as high a speed as possible up to the full expected speed of the IP in its final use in silicon. We would also prefer to make the prototype as accurate as possible and to interface with real-world peripherals. This is the use model for most of the prototyping performed within the Synopsys IP groups, for example in Figure 130 (as previously mentioned in chapter 2) from Synopsys IP design group in Porto, Portugal, we see the IP under test at the bottom of the FPGA block diagram while the rest of the FPGA is used to create a validation environment for the IP under test.

The channel from the HDMI Rx IC through the audio/visual processing out into an external PHY transmitter is what we want to prototype. However, we also need support elements for control and integration with the AV channel. This would be performed differently in each SoC implementation but for the purposes of running some software to test the HDMI channel, we pull together an infrastructure inside the FPGA using IP readily available from Xilinx and standard external memory components. The two halves are linked together using standard Synopsys IP for the ARM AMBA® interconnect.

Figure 130: FPGA IP augmenting a test rig for SoC IP



In this chapter, we shall focus not so much on this use mode as, in many ways, prototyping a specific piece of IP is really a subset of the tasks involved in SoC prototyping in general. In fact, IP prototyping is probably easier than for a whole SoC because many IP designs will fit into a single FPGA and thus we avoid partitioning. Let us move on to the more general case, then, of including peripheral IP into a larger SoC prototype.

10.4.2. Use mode 2: prototype the IP as part of an SoC

The second use model, and the one most relevant to this chapter, is the use of an IP within a larger SoC. In that case, although we would indeed prefer to run at full-silicon speed, we will probably find that our overall system speed is limited by the performance of the SoC RTL when mapped into the FPGA core. In that case our task becomes a matter of providing a method for scaling the overall system speed in order to more closely match the peripheral IP speed with the SoC core. This may involve data buffering or splitting an input stream into parallel processing channels.

10.4.2.1. Rate adapting between core and peripheral IO

What can we do when the peripheral data is arriving too fast for the FPGA to be able to handle? A common way to remedy clock differences between the FPGA system and an external interface is to add “rate adapter” circuits between the external stimuli and the SoC. These circuits are logically FIFOs that “absorb” (and drive) the external stimuli at the full external speed, although probably at its lowest acceptable rate to still meet the standard. The adaptor then only processes a subset of the stimuli that the FPGAs can process with its reduced clock rate, and it then drives the stimuli to the external interface at the reduced clock rate.

For example, if an FPGA-based prototype runs at one-third the speed of the SoC then, to properly emulate the SoC’s performance, the FPGAs should be driven by one-third the amount of stimuli.

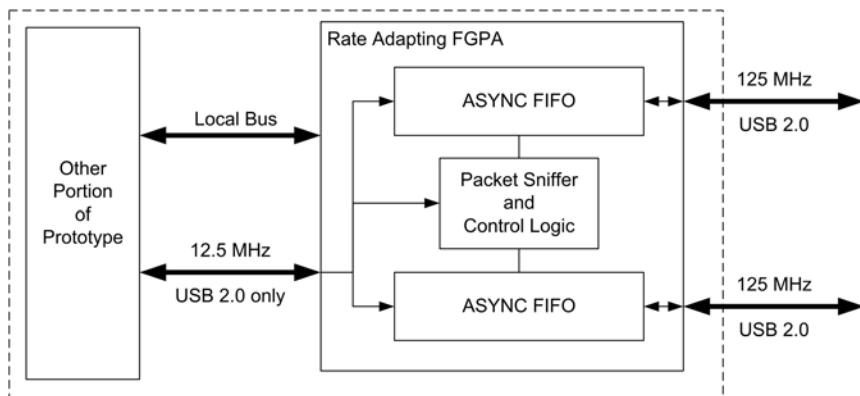
Another alternative is to predict the maximum amount of data that will arrive in a burst at the full peripheral rate. We can then implement a shift-register first-in-first-out (FIFO) buffer to receive it at full rate and then pass forward to the FPGA prototype at the reduced rate after all data is received. The same approach is used in the other direction to step up the rate from the FPGAs to the external interface. Some designs will not tolerate this approach but for many others this rate-adapting is adopted with great success.

The FIFO can be implemented in external components, in another FPGA, as in the example in Figure 131 or in the receiving FPGA itself, depending in speed and size requirement. In the example, designed by Synopsys consultants in New England, we

see two external USB2.0 channels being run at 125MHz, and asynchronous FIFOs being used in each case to buffer incoming and outgoing packets. Complete packets themselves are recognized by the “sniffer” circuit and signaled to the rest of the prototype as being ready to read in at one-tenth speed.

This kind of rate adaption works very well for regular packets of data and can handle reasonable rates as long as data is in bursts. It can also handle deep data if enough FIFO memory is available but, obviously, continuous data traffic would overflow the FIFO and packets would be lost or would need to be discarded until the rest of the prototype is ready to receive.

Figure 131: Rate adapter concept for 10x USB reduction



This area of prototype design requires some additional engineering beyond that already in the SoC, but many find that to be the most interesting part of a prototyping project. The best time to consider buffering between internal and external data streams is when the SoC is originally being designed. Having the prototyping team involved at this early stage will allow rate-scaling design to be completed in time and for software teams to pre-empt the necessary scaling changes in their code.

10.4.3. Use mode 3: prototype IP for software validation

The third common use case in Table 28 (on page 304) is when we want only to run applications software on a platform and our task is to provide data to the software and receive results as if we were running on the final system. From a software viewpoint we do not have to be cycle accurate as long as the channels run with “enough speed” to keep the channels in sync and provide approximately the required functionality. For example, as long as that the software thinks that it is

receiving video data from an HDMI source, then the transceiver channel for that video data might be quite different from that instantiated in the SoC design itself. In fact, we could even completely replace the SoC IP with an FPGA version. We could also consider replacing the HDMI channel with a transactor that performs the same job, as we shall see in chapter 13.

10.5. Use of external hard IP during prototyping

As mentioned in section 10.2, the only form of IP available in some cases is an external test chip or hard IP. There are many reasons why we would want to use a hard IP block alongside our FPGAs, some of which may be forced upon us, for example, the RTL is not available or not supported by the IP vendor for external usage.

The most common hard IP blocks found in prototypes are CPU cores and standard bus interfaces using PHY, including Ethernet, USB, PCI/PCI-X/PCIe and memory interfaces such as DDR2. Many vendors offer their IP as hard macros pre-implemented by various semiconductor partners and as end users we might choose a hard IP which most closely reflects our final silicon usage.

A good by product of using test chips and external hard IP is that they free-up FPGA resources. These are then freed up for other uses or left unused, lowering the device utilization and therefore decreasing runtime and potentially increasing performance. We should note that the FPGA resources required for large IP blocks such as DSPs or processor sub-systems could easily consume a whole FPGA or even multiple FPGAs. In the latter case the cost and performance impact of splitting IP across multiple FPGAs might be too high, especially if we do not have intimate visibility of the internal workings of the IP. In these cases, it would make sense to use an external hard IP or test chip, accepting the limitations this may impose compared to the effort required to develop a larger multi-FPGA solution. These are amongst the decisions to be made in the early stages of the project (see chapter 4).

10.6. Replacing IP or omitted structures with FPGA IP

There are situations where we may need to include logic modules in the design for which there is no RTL available, but the same functionality is available from another source. FPGA vendors, including Xilinx, offer a wide range of IP cores that can be placed into the design in place of the core instantiated in the SoC. These cores are often optimized for FPGA and should give better performance and area results than would have been achieved by simply porting the RTL of the original SoC IP. Note that these Xilinx® IP cores may not be used in the SoC (Xilinx only licenses its IP for use in its own devices).

Some examples for such IP are specialized memories, processors, communication controllers, multi-gigabit controllers and many more. The cores are in the form of a low-level FPGA netlist, either generated with a Xilinx supplied tool (CORE Generator tool) or from previous FPGA implementation. The IP is instantiated into the RTL source before synthesis which supports the inclusion of FPGA IP blocks in the following ways:

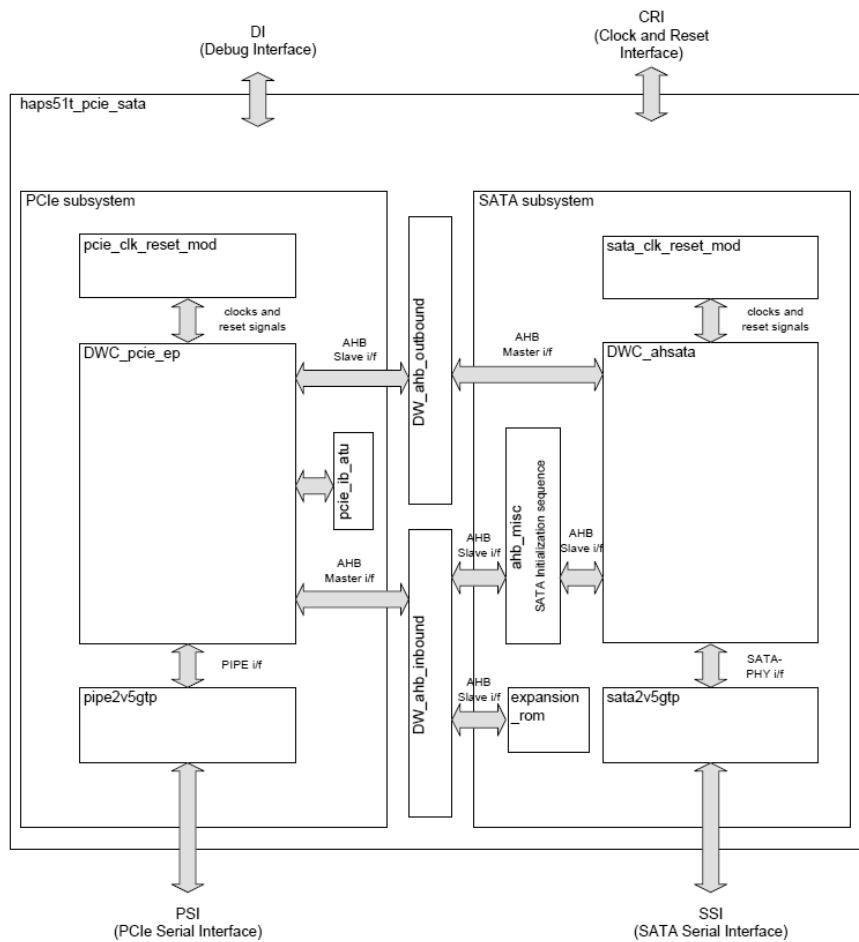
- **Plain text netlist:** the preferred method for including IP in the design, where synthesis reads the core's FPGA-level netlist, optimizes it as necessary and includes the results in the output netlist. It also passes down timing constraints to the place & route tools if supplied by the user.
- **White box:** usually used when secured IP is included in the design and synthesis reads the core's netlist to obtain timing information for the IP boundary, but does not optimize the core's logic.
- **Black box:** usually used when secured IP is included in the design, and the black box attribute is specified. Modules defined as black boxes are treated by synthesis as “space holders,” where it does not read the core's netlist (hence won't optimize it). The IP/core will be included in the design at the place & route stage, when the associated pre-generated netlist will fill in the “black box” space. The tool however will pass to the place & route timing constraints if specified by the user.

10.6.1. External peripheral IP example: PCIe and SATA

The example we shall use is a PCIe-to-SATA bridge that was developed at the Synopsys IP development lab near Dublin, Ireland. Synopsys IP teams have used FPGA-based prototyping extensively to validate IP and its connectivity with external systems. In order to do so, demonstration and early-adopter platforms are created which need to closely resemble as many of the eventual targets for the IP as possible. The references give more details but the block diagram of the design is shown in Figure 132.

This example used the Virtex[®]-5 LS330 device on a HAPS-51 board to implement both the PCIe and SATA interfaces and the bridge between them; another testimony to the scale of modern FPGA devices. The interfaces use the built-in fast serial transceivers on the FPGAs and have bespoke blocks to drive from the cores to the FPGA-specific hardware, we can see the special FPGA-ready modifications labeled as pipe2v5gtp and sata2v5gtp. Pipe is an intermediate interface between the PHY of a PCIe interface and the rest.

Figure 132: Top-level implementation of PCIe-to-SATA bridge design



This arrangement allowed the prototypers to fully test the two sub-systems with real world data and realistic speeds.

10.6.2. Note: speed issues, min-speed

In an ideal world we would want to be able to clock IP at any chosen speed, from DC to 1GHz+ to allow us to fully validate the systems features and enhance debug and development. However, we know that a crossover point normally occurs at about 20-100MHz when we need to migrate from a simple clocked design to

something that requires internal clock generation and de-skew logic as mentioned in chapter 8. This requires the use of PLL and MMCM type blocks but this will enforce a minimum operation speed on test chips and FPGAs alike. This is limited by the PLL/MMCM operating specification and it is risky to try to go slower than that, even if it may seem to work in the lab.

On another related note, the PLL in the SoC and the replacement in the FPGA may have different jitter, duty cycle, and even different frequencies of operation, drift, and accuracy. Therefore we need to be sure that the use of the MMCM or other clock circuitry is accurate and close enough to the SoC infrastructure give meaningful results. That is not usually in doubt for regular digital prototypes running within spec.

10.7. Summary

One of the biggest challenges for a new prototyping team is the inclusion of IP, especially high-speed peripheral IP. Third-party IPs are often provided without source code, so they cannot be synthesized as part of the RTL and an alternative must be found. This could be a netlist in the SoC library or pre-mapped to FPGA elements. It could also be an external testchip or an encrypted version of the RTL.

All of these forms of the IP can be used but we should not lose sight of the purpose which is not to verify the IP; that should have been guaranteed already. The reason to model the IP is so we do not leave a hole in the design. Having the IP present in some form allows us to validate the rest of the hardware and especially the software running upon it.

The authors gratefully acknowledge significant contribution to this chapter from

Spencer Saunders of ARM, Cambridge

Antonio Costa of Synopsys IP Group

Peter Gillen of Synopsys IP Group

Torrey Lewis of Synopsys IP Group

David Taylor of Xilinx, Edinburgh

We have come a long way. We have chosen or built our FPGA platform; we have manipulated our design to make it compatible with FPGA technology and we are ready with bitstreams to load into the devices on the board. Of course, we will find bugs, but how do we separate the real design bugs from those we may have accidentally introduced during FPGA-based prototyping? This chapter introduces a methodical approach to bringing up the board first, followed by the design, in order to remove the classic uncertainty between results and measurement.

We can then use instrumentation and other methods to gain visibility into the design and quickly proceed to using our proven and instrumented platform for debugging the design itself.

During debug we will need to find and fix bugs and then re-build the prototype in as short an iteration time as possible. We will therefore also explore in this chapter more powerful bus-based debug and board configuration tools and running incremental synthesis and place & route tools to save runtime.

11.1. Bring-up and debug—two separate steps?

It is worth reminding ourselves of the benefits of prototyping our designs on FPGA. We use prototypes in order to apply high-speed, real-world stimulus to a design, to verify its functionality and then to debug and correct the design as errors are discovered. The latter debug-and-correct loop is where the value of a prototyping project is realized and so we would prefer to spend the majority of our time there. It is tempting to jump straight to applying the whole FPGA-ready design into the prototyping platform, but this is often a mistake because there are many reasons why the design may not run first time, as will be discussed later in this chapter. It is very difficult in this situation to determine what prevents a design from running first time, so a more methodical approach is recommended.

When the design does not work on a prototyping board, it could be because of two broad reasons. It could be because of problems related to the prototyping board setup, or due to problems in the design that is run on the board. Separating the

debugging process for these two separate problems will lessen the whole debugging time.

For effective debug-and-correct activity it is critical to make sure that the bugs we see are real design issues and not manifestations of a faulty FPGA board or mistakes in the prototyping methodology. We therefore should bring up our design step-by-step in order to discover bugs in turn as we test first the board, then the methodology and finally the completed design in pieces and as a whole. This will take more time than rushing the whole design onto the boards, but in the long run will save time and help prevent wasted effort.

These bring-up steps can be summarized as follows:

- Test the base board
- Test the base plus the add-on boards
- Apply a small reference design to single FPGAs
- Apply reference design to multiple FPGAs
- Inspect SoC design for implementation issues
- Apply real design in functional subsets in turn
- Apply whole design

So, it is always necessary to make sure that the FPGA board setup is correct before testing the real design on board. The next step is to bring up the design on board by making sure that the clock and reset signals are correctly applied to the design. After the initial bring up, the actual design validation stage would start. In this design validation stage, debugging the issues becomes easy when there is enough visibility to the design internals. The necessary visibility can be brought into the design using different instrumentation methodologies which will be discussed in detail in the later part of this chapter.

11.2. Starting point: a fault-free board

We should like to start this chapter from the assumption that the FPGA board or system itself does not have any functional errors, but is that a safe assumption in real life? As mentioned in chapter 5, this book is not intended to be a manual on high-speed board design and debug so we are not intending on going into depth on board fault finding. We assume that those who have developed their own FPGA boards in house will know their boards very well and would have advanced to the point where the boards are provided, fully working to the lab.

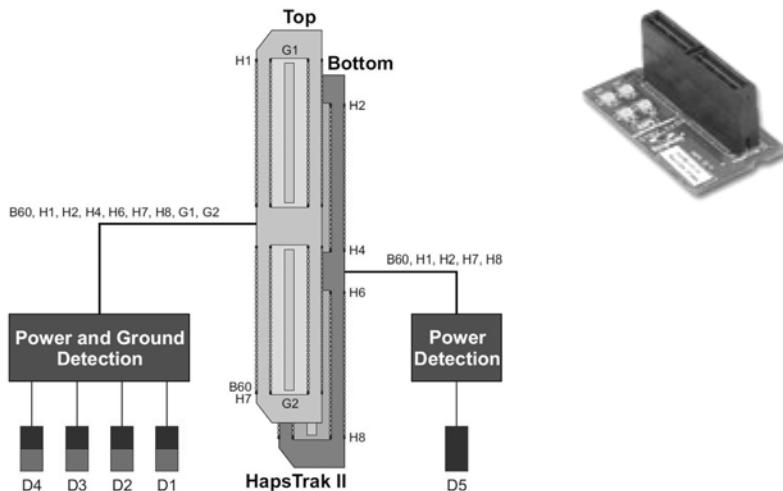
There is an advantage at this time for those using commercial boards because these would have already been through a full-production test and sign-off at the vendor's

factory. For added confidence, board vendors should also supply user-executable test utilities or equipment in order to repeat much of the factory testing in the lab. For example, a special termination connector or automatic test program to run either on the board or on a connected host. It is not paranoid to recommend that these bare-board tests are repeated from time-to-time in order to reinforce confidence, especially when the prototype is showing some new or unexpected behavior. A few minutes running self-test might save many hours of misdirected analysis.

As well as self-test, board vendors will often provide small test designs for checking different parts of the base boards. Even if in-house boards are in use, it is still advisable to create and collect a library of small test designs in order to help commission all the parts of the boards before starting the real design debugging work.

The kinds of tests applicable to a base board vary a great deal in their sophistication and scope. As an example of the low-end tests, the Synopsys® HAPS® boards have for many years employed a common interconnect system, called HapsTrak®. Each board is supplied with a termination block, called a STB2_1x1 and is pictured in Figure 133. The STB2_1x1 can be placed on each HapsTrak connector in turn in order to test for open and short circuits and signal continuity.

Figure 133 : STB2_1x1. An example of a test header for FPGA Boards



Testing the board from external connectors in this way would be a minimum requirement. Beyond this, we might also want to use any available built-in scan techniques, additional test-points, external host-based routines etc. All of these are vendor- and board-specific and apart from the simple above example, we leave it to the reader to explore the particular capability of their chosen platform. For the purpose of this manual, we will start from the previously mentioned assumption that

the board itself is functional before we introduce the design. However, there are other aspects of the board and lab set-up that may be novel and worth checking before the design itself is introduced.

Typical issues that would present in the prototyping board setup are problems in the main prototyping board and daughter boards themselves, connection between these different boards through connectors and cables, or issues related to external sources like clock and reset sources. These all come down to good lab practice and we do not intend to cover those in detail here.

11.3. Running test designs

The best way to check whether the FPGA board setup is correct is to run different test designs on the board and check the functionality. By running a familiar small design which has well-known results, the correct setup of the boards can be established. Experienced prototypers can often reuse small test designs in this way and in a matter of hours know that the boards are ready for the introduction of the real SoC design.

Board vendors may also be able to supply small designs for this purpose but in-house boards will require their own tests and be tested separately before delivery to the prototypers. Use of custom-written test designs to test these boards in the context of the overall prototype can then be performed. This also applies to custom daughter boards for mounting onto a vendor-supplied baseboard. For example, if a custom board is made to drive a TFT display then a simple design might be written to display some text or graphics on a connected display in order to test the custom board's basic functionality and its interface with the main board.

In either case, it is recommended to connect all the parts of the prototype platform including the main board, daughter boards, power supply, external clocks, resets, etc. as they will be used during the rest of the project. Then configure the assembled boards to the exact setting with which the actual design will employ during runtime, including any dip switches and jumpers. Sometimes these items can be controlled via a supervisory function which will access certain dedicated registers on the board under command of a remote-host program. It is especially important to configure any VCO or PLL settings and the voltage settings for each of the different IO banks of the FPGAs. Off-the-shelf prototyping boards like the HAPS board provide an easy way of configuring this using dip switches and on-board supervisory programs.

We are then ready to run some pre-design tests to ensure correct configuration and operation of the platform. Here are some typical test designs and procedures which the authors have seen run on various prototyping boards.

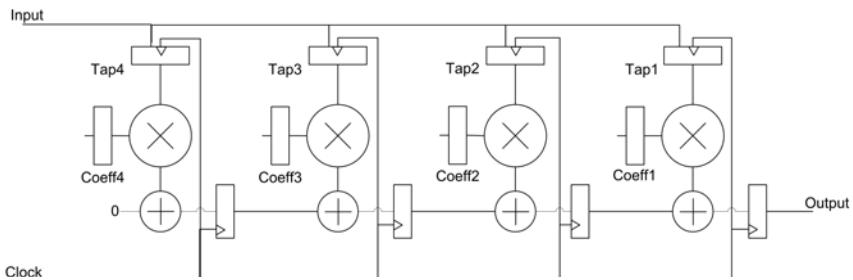
- **Signs-of-life tests** are very simple. Reading or writing to registers from an external port can confirm that the FPGA itself is configuring correctly.

- **Counter test** to test clocks and resets are connected and working correctly. Write a simple counter test-design with the clock and reset as inputs and connect the outputs to LEDs or FPGA pins which connect to test points in the board, or read the counter register using an external host program.
- **Daughter board reference designs** provided by vendors or equivalent user-written test designs can be used to check the functionality of the add-on daughter boards. Testing with these designs would test the proper functionality of the daughter boards and the interface between the main board and the daughter boards and links to external ports, such as network analyzers.
- **High-speed IO test:** If the prototyping project makes use of advanced inter FPGA connectivity features like pin multiplexing or high-speed time division multiplexing (HSTDMD), it is advisable to run simple test designs to test this on the board. Owing to the environmental dependency of LVDS and other high-speed serial media the test designs should employ the same physical connections in order to properly replicate the final paths to be used in the SoC prototype.
- **Multi-FPGA test designs:** If the design is partitioned across multiple FPGAs, then it is advisable to test the prototyping board with a simple test design which is easily partitioned across all the devices. One example of this would be an FIR filter with each tap partitioned in a different FPGA. Let's look at that example in a little more detail.

11.3.1. Filter test design for multiple FPGAs

One example of a simple-to-partition design which tests many aspects of a multi-FPGA design configuration is the FIR shown in Figure 134 which would serve to test four FPGAs on a board.

Figure 134 : A 4-tap pre-loadable transposed FIR filter test design



An FIR, being a fully synchronous single-clock design helps to check that reset timing is correct and that clock skew is within an acceptable range.

Figure 135 : RTL code for FIR filter test design

```
module TransposedFIRFilter (
    input Clock, Reset,
    input signed [17:0] Input,
    output reg signed [35:0] Output
);

reg signed [17:0] InputReg1, InputReg2, InputReg3, InputReg4;
reg signed [35:0] TempOutput2, TempOutput3, TempOutput4;
parameter signed
    Coeff1 = 18'd87256,
    Coeff2 = 18'd1256,
    Coeff3 = 18'd7344,
    Coeff4 = 18'd32353;

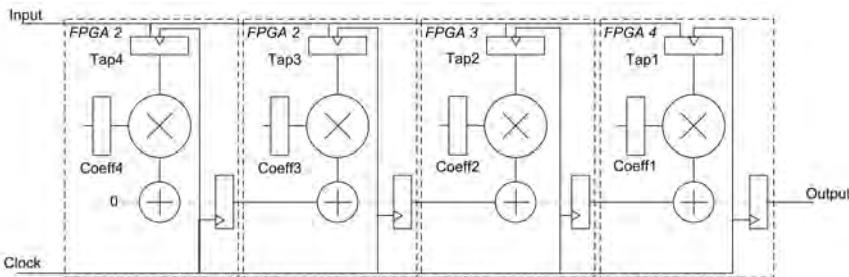
always@(posedge Clock) begin
    if(Reset == 1'b1) begin
        InputReg1<= 0; InputReg2<= 0; InputReg3<= 0; InputReg
4<= 0;
        end
    else begin
        InputReg1 <= Input; InputReg2 <= Input; InputReg3 <=
Input; InputReg4 <= Input;
        end
    end

always@(posedge Clock) begin
    if(Reset == 1'b1) begin
        TempOutput4 <= 0; TempOutput3 <= 0; TempOutput2 <= 0;
Output <= 0;
        end
    else begin
        TempOutput4 <= InputReg4*Coeff4 + 0;
        TempOutput3 <= InputReg3*Coeff3 + TempOutput4;
        TempOutput2 <= InputReg2*Coeff2 + TempOutput3;
        Output <= InputReg1*Coeff1 + TempOutput2;
        end
    end
endmodule
```

The RTL for this design is seen in Figure 135 and this may be synthesized and then split across four FPGAs either manually or by using one of the automated partitioning methods discussed earlier. The values of the coefficients, although arbitrary, should all be different.

The design is partitioned so that each tap (i.e., multiplier with its coefficient and adder) is partitioned into its own FPGA, as shown in Figure 136. Care should be

Figure 136 : FIR filter test design partitioned across 4 FPGAs



taken with FPGA pinout and use of on-board traces to ensure that connectivity between the FPGAs is maintained.

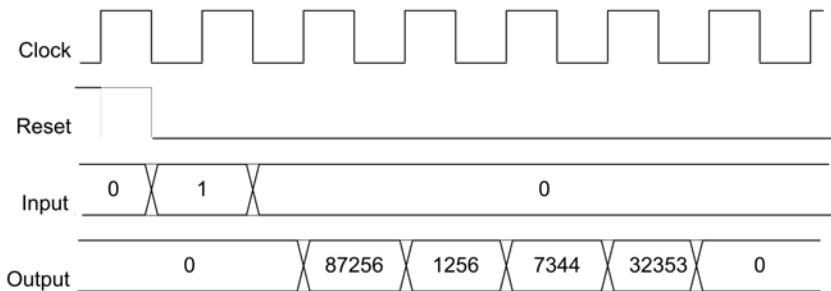
A logic analyzer is connected at the output and a pattern generator is connected at the input. Once the design is configured into the FPGAs then it can be tested by applying an impulse input as shown in Figure 137. The pattern generator can be used to apply the impulse input. An impulse input consists of a "1" sample followed by many "0" samples. i.e., for 8-bit positive integer values, then the input would be 01h for one clock followed by 00h for every clock thereafter. As an alternative to using an external pattern generator, a synthesizable “pattern generator” can be designed in one of the FPGA partitions to apply the impulse input. This is a simple piece of logic to provide a logic ‘1’ on the least significant bit of the input bus for a single clock period after reset.

The expected output from FPGA4 should be the same as the chosen filter coefficients and they should appear sequentially at FPGA4’s output pins as the pulse has been clocked through the filter (i.e., one clock after impulse input has applied in this 4-tap example), as shown in Figure 137.

11.3.2. Building a library of bring-up test designs

As the team becomes more familiar with FPGA-based prototyping, common test designs will be reused for different prototypes. It is advisable to create a way to share test designs amongst team members and across different projects. Creating a reusable test infrastructure with a library of known good tests usable on different boards is an investment that will benefit multiple prototyping projects and increase efficiency of our FPGA-based prototyping. This library could be considered in advance, or gradually built up with each new prototyping project.

Figure 137 : Expected FIR behavior across four FPGAs



By running such simple designs on the prototyping board setup, the whole setup is tested for its basic functionality and we get comfortable with the FPGA-based prototyping flow. After running few designs on the board, we would know how to partition a design, synthesize and place & route them, create bit files, program the FPGAs with the bit files and do some basic debugging on the board. This knowledge would help us while debugging the real design on the board setup.

By testing this design on board, we become familiar with the complete flow of partitioning a design across multiple FPGAs, managing the connections between them and running them on board. In fact, during early adoption of FPGA-based prototyping by any project team, this type of simple design is good for pipe-cleaning methods to be used in the whole SoC design later.

11.4. Ready to go on board?

At this stage, some teams will decide to introduce the SoC design onto the FPGA board. It may have seemed like a long time to finally reach this stage but an experienced prototyping team will perform these test steps discussed in section 10.1 in a few hours or days at most. As mentioned, a piecemeal approach to bring-up may save us a great deal of false debug effort.

There is one further step which is recommended for first-time prototypers or any team using a new implementation flow and that is to inspect the output of the implementation flow back in the original verification environment. Our SoC design has probably undergone a number of changes in order to make it FPGA-ready, not least, partitioning into multiple devices. There may be other more subtle changes that may have crept in during the kind of tasks described in chapter 7 of this book. How can we check that the design is still functionally the same as the original SoC,

or at least only includes *intentional* changes? The answer is to reuse the verification environment already in use for the SoC design itself.

11.4.1. Reuse the SoC verification environment

As mentioned in chapter 4, the RTL should have been verified to an acceptable degree using the simulation methods and signed off as ready for prototyping by the RTL verification team. Their “Hello World” test will probably have already been run upon the RTL and it is very helpful if this same test harness can be reused on the FPGA-ready version of the design.

We have seen that by reusing the existing SoC test bench we can identify differences between the behavior of the design before and after the design is made ready for the prototype. It is important that any functional differences between the SoC RTL and the FPGA-ready version can be accounted for as either intentional or expected. Unexpected differences should be investigated as they may have been introduced while we were preparing the prototype, often in the implementation process.

11.4.2. Common FPGA implementation issues

Despite all our best efforts, initial FPGA implementations can be different to the intended implementation. The following list describes common issues with initial FPGA implementations:

- **Timing violations:** timing analysis built into synthesis and place & route tools operates upon each FPGA in isolation. Timing violations across multiple FPGAs, for example, on paths routed through FPGAs or between clock domains in different FPGAs, would not be highlighted during normal synthesis and place & route.
- **Unintended logic removal:** it may not be obvious at first, but modules and IO seem to “disappear” from the resulting FPGA implementation due to minimization during synthesis. The common cause is improper connectivity or improper modules and core instantiations that result in un-driven logic, which is subsequently minimized. Early detection of accidental logic removal can save valuable FPGA implementation time and bench debug time. A review of warnings and the FPGA resource utilization, especially IO, after design completion will indicate unintended logic removal, for example if there is a sudden unexplained drop in IO.
- **Improper inter-FPGA connectivity:** despite all efforts, due to improper pin location constraints, the place & route process will assign IO to unintended pins resulting in unintended and incorrect inter-FPGA

connectivity. This often happens the first time a design is implemented or drastically modified. The remedy for this issue is to carefully examine the pin location report generated by the place & route tools for each FPGA and compare against the intended pin assignments.

Let's consider each of these types of implementation issues in more detail.

11.4.3. Timing violations

The design may have timing violations within an FPGA or between FPGAs. It is always advisable to run the static timing analysis with proper constraints on the post place & route netlist and make sure that all timing constraints are met. Even though FPGA timing reports consider worst case process, voltage and temperature (PVT) conditions, if timing constraints as reported by the timing analysis are not met, there is no guarantee the design will work on board.

Here are some common timing issues we might face while analyzing timing on a design and some tips on how to handle them:

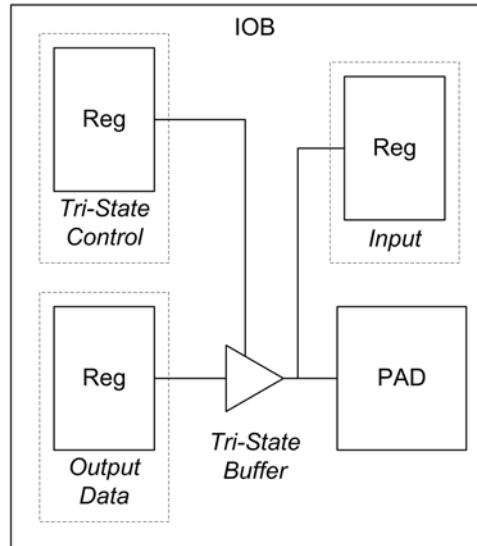
- **Internal hold time:** if we are prototyping the design at a very slow clock rate then we might be tempted to think that there cannot be timing violations on today's fast FPGAs. As a result we might not choose to run full timing analysis. However even in such scenarios, there can be hold time violations which will prevent the design from working on board. So, even if the timing requirements are very relaxed, FPGA internal timing should be analyzed with appropriate constraints and with the use of *minimum* timing models.
- **Inter-FPGA delays:** careful analysis of inter-FPGA timing, taking in account board delays, should be made to ensure inter-FPGA setup and hold times are met. For this we need to know the appropriate board and cable delays to account for these either within the timing model or by setting appropriate constraints for whole-board timing analysis. If off-the-shelf standard prototyping boards and their associated standard cables are used for inter-FPGA connection then the expected board and cable delays should be available from the vendors. For example, the Synopsys HAPS series boards are designed to have track delays matched across the boards and between boards and delays are specified in terms of two constants, X and Y. We referred to these X and Y delay specs in the PLL discussion in section 5.3.1 and for a HAPS-54 boards for example the nominal values are X=0.44ns and Y=1.45ns. These values can be used to add delays into calculations for inter-FPGA delays.
- It is an advantage if the boards are pre-characterized for use in timing analysis tools, as is the case for HAPS boards in the Certify® built-in timing analyzer. If a custom-made board or manual partitioning is used

then whole-board timing analysis would be more complex but could still be done as long-delay data could be provided by the PCB development and fabrication tools.

- **Timing complexity of TDM:** If the prototype uses pin multiplexing to share one connection between FPGAs for carrying multiple signals, then the effect of this optimization on the inter-FPGA timing should be analyzed carefully. See chapter 7 for more consideration of the timing effects of signal multiplexing.
- **Input and output timing:** constraints on all the FPGA ports through which the design interacts with the external world should not be neglected. Without proper IO constraints, the interface with the external peripherals may not work reliably. Timing is easier to meet if the local FF features of the FPGA's IO buffers are used to their full extent, thus removing a potentially long routing delay from an internal FF to reach the IO Blocks (IOBs). As was shown in chapter 3, all the IOBs of the FPGAs have dedicated FFs for data input, output and tri-state enables. These FFs should be used by default by the synthesis and place & route tools, but may require some intervention using the tool-specific attributes.

If the FFs have not been used properly during the implementation flow then timing problems may be introduced. For example, consider the simplified view of a typical FPGA IOB shown in Figure 138 and its use as

Figure 138 : Typical FF arrangement in an FPGA IO cell



a tri-state output pin. The recommendation is to use the FFs available in IOB for both the tristate control path and the output datapath. For tristate signals, timing advantage of using the IOB FF can be realized by using both the tri-state FF and output data FF. If neither FF is used then the extra routing delay in that path will nullify the timing advantage obtained in the other path.

For example, if the output path uses its IOB FF and the tristate control path does not use its FF then the output data may indeed arrive sooner at the tri-state buffer. The output data would only reach the PAD after the tri-state control reaches the tri-state buffer and after that buffer's switch-on delay. In effect, the timing advantage produced by using the IOB FF in the datapath is offset by the non-optimal delay in the tri-state control path.

Similarly, if the tri-state control path uses the IOB FF and output datapath does not use its FF then the tri-state buffer will put the previous output FF data at the output PAD when the tri-state control is asserted and after a little while the new data will arrive at the output PAD. This skew in the arrival time of tri-state control and data may introduce glitches at the PAD output which may or may not be tolerable at the external destination.

- **Inter-clock timing:** as is true for all logic design, we should take special care with multi-clock systems when signals traverse between FFs running in different clock domains. If the two clocks are asynchronous to each other then there can be setup and hold issues leading to metastability (check references for more background on metastability). Avoiding metastability between domains is as much a problem in FPGA as it is in SoC design so similar care should be taken. In fact, the measures taken in the SoC RTL to avoid or tolerate metastability (e.g., double-latching using two FFs in series on the receiving clock) can be transferred directly into the FPGA but we should also apply all the timing constraints used for ASIC to the FPGA.
- We can ensure that the probability of meta-stable states is within reason or otherwise take counter-measures. In either case we should reassure ourselves that the problem is under control before going onto the board. Timing analysis for each FPGA may indicate where metastability can occur. For example, Synopsys FPGA synthesis tools generate specific warning messages for signals traversing clock domain boundaries and these messages can be checked manually or by a script. This becomes more complex when analyzing multiple FPGAs. One suggestion is to avoid setting partition boundaries so that the sending FPGA and receiving FPGA are on different asynchronous clocks.
- **Gated clock timing:** if there are gated clocks in the design which are not converted then there is potential for hold-time violations to occur inside the FPGAs, caused by clock skew and possibly even glitches on poorly

constructed clock gates (although the latter is a real bug and good to feed back to the SoC team). Preferably all gated clocks will have been converted using techniques discussed in chapter 7, but it is possible that some have been omitted or not converted correctly leading to timing issues after place & route. Close inspection of the synthesis tools report files for messages regarding the success or failure of gated-clock conversion can often give clues to the cause of unexpected timing violations.

- **Internal clock skew and glitches:** we should be using the built-in global and regional clock networks which are inherently glitch free. Check for low-fanout clocks that have been accidentally mapped to non-global resources.
- **Timing on multiplexed interconnect:** as we saw in chapter 8, the timing of TDM connections between FPGAs can be crucial. We need to ensure that the timing constraints on both the design clock and the transfer clock are correct and confirm that they are met by performing the post place & route timing analysis. It is especially important to re-confirm that the time delay between design and transfer clock is within limits, that the on-board flight time is not longer than expected, and that, if asynchronous TDM is in use, synchronization has been included between the design and transfer clock domains.

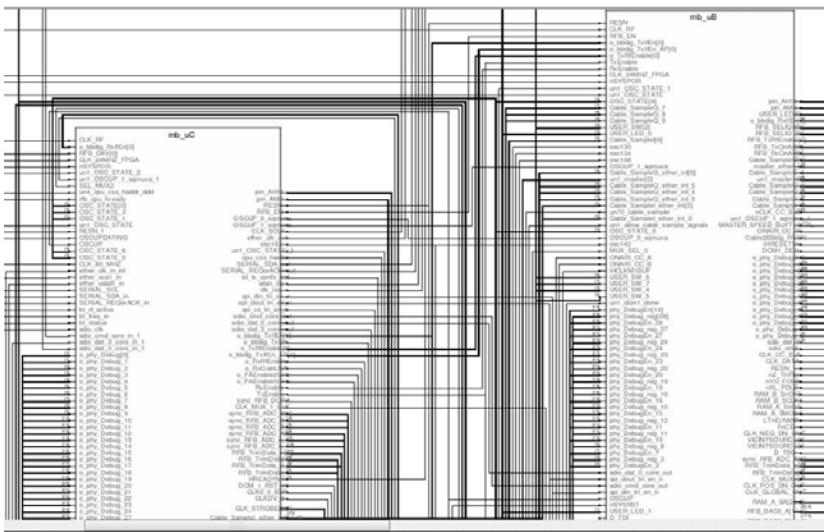
We can see from these examples above that thorough timing analysis of the whole board is very useful and can discover many possible causes of non-operation of a prototype before we get to the stage of introducing the design onto the board. Any of these timing problems might manifest themselves obscurely on the board itself and potentially take days to uncover in the lab.

11.4.4. Improper inter-FPGA connectivity

Another difficult-to-find problem is improper connectivity between the FPGAs on the board. That is, signals between FPGAs which are misplaced so that the source and destination pins are not on the same board trace. Keeping correct contiguity between FPGA pins should be a matter of disciplined specification and automation. However, as the excerpt from a top-level partition view in Figure 139 shows us (or rather scares us), there will be thousands of signal-carrying pins on the FPGAs in a typical sized SoC prototype. If any one of these pins is misplaced (i.e., the signal is placed on the wrong pin) then the design will probably not behave as expected in the lab and the reason might prove very hard to find.

If this happens then it is most likely during a design's first implementation pass or after major RTL modification or addition of new blocks of RTL at the top level.

Figure 139 : Certify® Partition View illustrating complexity of signals between FPGAs



Pin location constraints are passed through different tools in the flow, usually from the partition tool to the synthesis tool to the place & route tool. If any of the different tools in the flow drops a pin location constraint for some reason then the place & route tool will randomly assign a pin location with only low probability that it will be in the correct place. The idea of pin placements being “lost” might seem unlikely but when a design stops working, especially after only a small design change, then this is a good place to look first.

Possible reasons for pin misplacement include accidentally setting a tool control to not forward annotate location constraints. Another mistake is to rely on defaults that change from time-to-time or over tool revisions. A further example happens when we add some new IO functionality but neglect to add all of the extra pin locations. In every case, however, we will notice the missing pin constraints if we look at the relevant reports.

We should carefully examine the pin location report generated by the place & route tools for each FPGA and compare against the intended pin assignments. In the case of the Xilinx® place & route tools, this is called a PAD report. Human inspection of such reports, which can be many thousands of lines long, might eventually lead to an error so some scripting is recommended. The script would open and read the pin location reports and look for messages of unplanned pins but while doing so, could also be looking for other key messages, for example, missed timing constraints or over-used resources. In the case of the pin locations, we might maintain a “golden” pin location reference for all the FPGAs so that the script can automatically

compare this against the created PAD reports after each design iteration. This would add an extra layer of confidence on top of looking for the relevant “missing constraint” message. In addition, we could set up other scripted checks in order to compare any pin location information at intermediate steps in the flow, for example, between the location constraints passed forward by partition/synthesis to the place & route, and those present in the final PAD files.

It is worth noting that if a commercial partitioning tool is used then inter-FPGA pin location constraints should be generated automatically and consistently by the tool, perhaps on top of any user-specific locations. In the case of Certify, if the connectivity in the raw board description files are correct and the assignments of all the inter FPGA signals to the board traces are complete then there is a very minimal risk that the pin locations will be lost or misplaced by the end of the flow.

On the other hand, if we manually partition the design, then we need to carefully assign the pin location constraints for all the individual inter-FPGA connections as well as make sure that these are propagated correctly to the place & route stage, which can become a tedious and therefore error-prone task if not scripted or otherwise automated.

If all this attention to setting and checking pin locations seems rather paranoid then it is worth remembering that there might be thousands of signals crossing between FPGAs. In the final SoC, these signals will be buried within the SoC design and the verification team will go to great lengths to ensure continuity both in the netlist and in the actual physical layout. A single pin location misplacement on one FPGA would be as damaging as a single broken piece of metallization in an SoC device and perhaps harder to find (albeit easier to fix). Therefore it pays to be confident of our pin locations before we load the design into the FPGAs on the board.

11.4.5. Improper connectivity to the outside world

As well as between FPGAs, we need the proper connectivity between the FPGAs and any external interfaces. Some of the simplest but most crucial external interfaces are clock and reset sources, configuration ports and debug ports. The more sophisticated connections would be interfaces to external components like DDR SDRAM, FLASH and other external interfaces.

Once again, the implementation tools rely on there being correct and complete information about physical on-board traces between the FPGAs and from the FPGAs to the external daughter cards which may house the DDR etc. On a modular system, with different boards being connected together, the board description will be a hierarchy of smaller descriptions of the sub-systems and with a little care, we can easily ensure that the hierarchy is consistent and that the sub-boards are themselves correctly described. It may be worthwhile to methodically step through the board description and compare it to the physical connection of the boards,

however, some systems will also allow this to be done automatically using scan techniques to interrogate each device on the boundary scan chain and check that they appear where they should, according to the board description.

A commercial board vendor will be able to provide complete connection information for the FPGA boards and for their connection to daughter boards for this purpose. In such a board description, inter-FPGA connections on the board will be labeled generically because they might conceivably carry any signal (or signals) in the design.

For connections to dedicated external interfaces, however, the signal on each connection is probably fixed e.g., a design signal called “acknowledge” must go to the “ack” pin on the daughter card’s test chip. In those cases, it can help to also give the traces in the board description the same meaningful name to make it easier to verify the connectivity to external interfaces and match signals to traces. So for example, we can easily check that a trace called “ack” on the board is connected to a pin called “ack” on the daughter card and is carrying a signal called “ack” from the design. This naming discipline is especially useful for designs with wide bus signals.

Some advanced partitioning tools will be able to recognize that signals and traces, or signals and daughter-card pins, have the same name and therefore quickly make an automatic assignment of the signal to the correct trace, which if the board description is correct, will automatically also assign the signal to the correct FPGA pin(s). All of our pin assignments would therefore be correct by construction.

11.4.6. Incorrect FPGA IO pad configuration

When connecting various components at the board level, care must be paid to the logic levels of all interconnecting signals and be sure they are all compatible at the interface points. As well as having the correct physical connection, the inter-FPGA signals and those between the FPGAs and other components must be swinging between the correct voltage levels. As we saw in chapter 3, FPGA cores run at a common internal voltage but their IO pins have great flexibility and can support multiple voltage standards. The required IO voltage is configured at each FPGA pin (or usually for banks of adjacent pins) to operate with required IO standards and is controlled by applying the correct property or attribute during synthesis or place & route.

This degree of flexibility must be controlled because a pin driving to one voltage standard may be misinterpreted if interfacing with a pin set to receive in a different standard. This may seem obvious but as with the physical connection of the pins, the scale of dealing with voltage standards on thousands of signals adds its own challenge. For all the inter-FPGA connections, the IO standards of the driving FPGA pin and the driven FPGA pin should be the same.

The partition tools should automatically take care of assigning the same IO standards for connected pins, or warning when they are not. The default IO standard for the synthesis and place & route may also suffice for inter-FPGA connections but it is not recommended to rely only on default operation. In addition, care should be taken when the design is manually partitioned.

The board-level environment may also constrain the voltage requirement and we will need to move away from the default IO voltage settings. Then there are special considerations for differential signals compared with single-ended.

Let's look at some of these board-level voltage issues here:

- **The correct IO standard for differential signals:** on a prototyping board, it is likely that only a subset of traces can be used in differential mode. Not only must pin location constraints be correct to bring signals to the correct FPGA pins to use these traces but also the pin's IO standard must be set correctly. It is a subtle mistake to have a differential signal standard at one end and single-ended standard at the other, so that even though the pins are physically connected, the signal will not pass correctly between FPGAs.
- **Voltage requirements for peripherals:** for all the connections with external chips and IOs, we should first identify the IO standard of the pins of the external chips and IOs and then apply the same IO standards for the corresponding FPGA pin connected.
- **IO voltage supply to FPGA:** while configuring a certain IO standard for a certain pin in an FPGA, we should connect the necessary voltage source to the VCCO pins and VREF pins of the FPGA's corresponding IO bank. As discussed in chapters 5 and 6, the boards must have the flexibility to be able to switch different voltage supplies to different banks. We must then make sure that proper supply voltages are connected to all the VCCO pins according to the chosen IO standards. This is usually a task of setting jumpers or switches or, in some cases, of using a supervisor program to control on-board programmable switches, as seen in chapter 5.
- **Termination settings:** some of the IO standards require appropriate termination impedance at the transmitting and receiving ends. The FPGA's IO pads can be configured for different kinds of terminations and different impedances, for example using the digitally controlled impedance (DCI) feature in Xilinx® FPGAs. Once again, it is worth a quick check at the end of the flow to make sure that these are configured as expected.
- **Drive current:** the ports in the FPGA which interact with external chipsets should be able to source or sink the required amount of drive current as specified in the data sheets of the external chipsets. The FPGA's IOs can be configured to source and sink different currents, for example on normal LVCMOS pins on a Virtex®-6 FPGA, this can be programmed to be between 2mA and 24mA. This would have already been considered during

the early stages of the prototyping project but it is important that the FPGA pins be configured to supply the current or else inconsistent performance will result. This is one of those user-errors that can remain hidden for much of the project as in lab conditions we might be lucky while the peripheral is not running at full spec or otherwise does not demand full current from the FPGA pin. However, at other times the behavior of the design might become inconsistent for no apparent reason because the software might be using a feature of the peripheral not previously enabled.

- **Poor signal integrity:** commercial prototyping boards typically have acceptable point-to-point signal integrity at the board level. the same board design will probably have been previously used for multiple designs worldwide. However, first time we use a custom-built boards we may need some careful analysis and debug for such issues. Even off-the-shelf systems can exhibit poor signal quality with “exotic” connectivity, for example, when creating buses shared by multiple FPGAs or cables that are too long or not properly terminated. It is always best to identify and fix the root causes when possible before programming the board with the real DUT. For marginal noise or signal quality issues, we can sometimes change the programmable slew rate and drive strength at the FPGA IO pin.

11.4.6.1. NOTE: run scripts to check IO consistency

Although most of the inter-FPGA IO considerations above will be managed by the partitioning tools and therefore consistent by construction, we should maintain a “golden” reference for IO standard, voltage, drive current and placement for the critical pins on the FPGA and certainly between the FPGAs and external peripherals. We can then compare the golden reference against the created PAD report for each FPGA, however, it is wrong to have to make repetitive manual checks on every iteration of the design, so it is worth taking the time to script these kinds of checks.

We have mentioned how scripts can be used to automate these post-implementation, pre-board checks. Tools will have their own commands for generating reports on a large number of details, including top-level ports. A script can make multiple checks on such reports in the same pass, for example verifying that the IO standards could be combined with the pin location constraints, checking that every top-level port or partition boundary signal has an equivalent entry in the FPGA location constraints. Automatically running these scripts in a larger makefile process will prevent running on into long tool passes using data that is incomplete, and wasting a lot of time as a result.

11.5. Introducing the design onto the board

Once we are confident that the platforms/boards are functional and we are confident of the results of our implementation flow then we can focus on introducing the FPGA-ready version of the SoC design onto the board.

This first introduction should be done in stages because it is unlikely that every aspect of the design's configuration will be correct first time. Some configuration errors have an effect of masking others and so bringing up the prototype in stages helps to reduce the impact of such masking errors.

After checking the prototyping board setup with some test designs and verifying that there are no FPGA implementation issues, the actual design to be prototyped can be taken to the board. Here are some useful steps to follow:

- Confirm that the prototyping board setup is properly powered up before programming the FPGAs. Then we can program all the FPGAs with the corresponding generated bit files. Please refer to FPGA configuration in the chapter 3 and external references noted in the appendices for further details on configuring the FPGAs. There are dedicated “DONE” pins in all Xilinx® FPGAs which get asserted when the FPGAs are programmed successfully. In off-the-shelf prototyping boards like HAPS boards, these DONE pins are connected to LEDs on the boards so that the LEDs glow when the FPGAs are programmed successfully. It is advisable to check and make sure that all these LEDs glow to indicate the success of programming all the FPGAs. During configuration, some FPGAs can draw their peak current, owing to the large number of elements switching inside the device fabric. This peak may be high enough to overload the board’s power supply. This is not to say the power distribution to the devices which should have been considered by the board developers long before its use in the lab. A more common power supply error during configuration is to have current limit on the lab power source set too low, leading to voltage rail brown-out during configuration.
- After programming the FPGAs and releasing the reset, it is worth doing a simple check to see that voltage rails are sound while the FPGAs are running. Some power supplies can power the bare prototyping boards but not when a real design is running on the FPGAs. Some commercial prototyping boards such as HAPS have built-in “power good” indicators and voltage sensing components which can be read by a supervisory program to help in this task.
- Initial checks can be performed at the inputs and outputs of the FPGAs and external components. Checking the inputs first makes sense and if these are valid, then move on to the FPGA outputs. Some of the inputs which need to be checked first are the clock and reset inputs, especially those which

are derived inside other the FPGAs. The inputs which are driven by any external chipsets should also be checked. Sometimes there are circular dependencies which can prevent start-up of the SoC design in FPGA. For example, the reset for one FPGA is driven by logic in another which depends upon a signal driven from the first FPGA.

- If all the inputs are valid then we can check all the outputs from the FPGAs and this leads us onto the subject of probing the FPGA. This is covered in depth in the next section.
- If the design spans multiple FPGAs, then start the debugging by checking the functionality of the parts of the design which reside in a single FPGA. This will eliminate the possibilities of problems in the inter-FPGA connections, complex pin multiplexing schemes, cables, connectors and traces on the prototyping boards.
- If the design has lot of interactions with external chip sets and IOs, then test all the parts of the design which reside only inside the FPGAs initially. This will eliminate the possibilities of problems in the external chipsets and problems occurring in the interactions.

11.5.1. Note: incorrect startup state for multiple FPGAs

A common issue with multi-FPGA systems is the improper startup state. In particular, if the FPGAs do not all start operating at the same time, even if they are only one clock edge apart, then the design is very unlikely to work.

Consider the simple case of a register-to-register path sharing a common clock. If the source and destination registers are partitioned into separate FPGAs and the receiving FPGA becomes operational just one clock cycle after the sending FPGA, then the first register-to-register transmission will be lost. Conversely, if the sending FPGA becomes operational one clock later than the receiving register, then the first register-to-register data may be random and yet still be clocked through the receiving FPGA as valid data. This simple example shows how there might be coherency problems across FPGA boundaries right from the start. These startup problems can also occur between FPGAs and external components such as synchronous memories or sequenced IOs.

A number of factors determine the time it takes for an FPGA to become operational after power up or after an external hard reset is applied, and it can vary from one FPGA to another. Prime examples are clocking circuitry (PLL/DCM) that use an internal feedback circuit and can take a variable amount of time to lock and provide a stable clock at the desired frequency. In addition, any hard IO such as Ethernet PHY or other static hardware typically becomes operational much sooner than the FPGAs as they do not need to be configured.

When the design shows signs of life but not as we know it, then one place to look is at this startup synchronization.

The remedy for the synchronized startup condition issue is a global reset management, where all system-wide “ready” conditions are brought to a single point and are used to generate a global reset. This is covered in detail in chapter 8.

11.6. Debugging on-board issues

After giving ourselves confidence that the FPGA board is fully functional and configured correctly, and also having checked the implementation and timing reports for common errors, we can consider that our platform is implemented correctly. From here on, any functional faults in the operation of the design will probably be bugs in the design itself that are becoming visible for the first time. The prototype is starting to pay us back for all our hard work.

The severity and number of the bugs discovered will depend upon the maturity of the RTL and how much verification has already been performed upon it. We shall explore in the remainder of this chapter some common sources of faults.

11.6.1. Sources of faults

Every design is different and we cannot hope to offer advice in this manual on which parts of the design to test first or which priority to place on their debug. However, we can offer some guidance on ways to gain visibility into the behavior and some often-seen on-board problems.

Assuming the design was well verified before its release to use, we should be looking for new faults to become evident because the design on the bench is exposed to new stimulus, not previously provided by the testbench during simulation. The cause of such faults can be found in three main areas, as listed in Table 29.

Table 29: Three main kinds of SoC bug discovered by FPGA-based Prototyping

1.	Logic bugs	Undiscovered RTL errors in the SoC design (e.g., real world data-dependency)
2.	Interface bugs	Unforeseen external interface issues (e.g., drive, standards)
3.	Software bugs	First seen during software-hardware integration (e.g., hard-to-find bugs in the software)

Type 1 bugs should be caught by the normal RTL verification plan and as mentioned previously, an FPGA-based prototype is not the most efficient way to discover RTL bugs, especially when compared to an advanced verification methodology like VMM. Nevertheless, RTL bugs do creep into the prototype either because there is an unforeseen error exposed by real-world data or because the RTL issued to the prototyping team is not fully tested.

Type 2 bugs are related to the SoC's interface with external hardware. All SoC's are specified to be used in a final product or system but there is a diminishing return on making specifications ultra-complete. Eventually there may be some combination of external components that do not fit within the spec and the design needs to be altered to compensate. This is often the case when extensive use of IP means that the SoC is the first platform in which a certain combination of IP has ever been used together. Alternatively, the design itself may be an IP block that is being specified to run in a number of different SoC designs for different end-users. Predicting every possible use of the IP is hard and corner cases are often discovered during the prototyping stage.

Type 3 bugs are the most valuable for the prototyping team to find. The prototype is the first place where the majority of the embedded software runs on the hardware. The interface between software and hardware is very complex having time dependency, data dependency and environmental dependency. These dependencies are difficult to model properly in the normal software development and validation flows, therefore on introduction to real hardware running at (or near) real-speed, the software will tend to display a whole new set of behaviors.

Any particular fault may be a combination of any two or indeed all three types of bug, so where do we start in debugging the source of a fault?

11.6.2. Logical design issues

Assuming that our newly found bug is not a known RTL problem, already discovered by the verification team since delivering the RTL for our prototype (always worth checking), we now need to capture the bug. We need to trace enough of the bug-induced behavior in order to inform the SoC designers of the problem and guide their analysis. This means progressively zeroing-in on the fault to isolate it in an efficient and compact form for analysis, away from the prototype if necessary.

The first step in identifying the fault is to isolate it to the FPGA level. To accomplish this, we need visibility into the design as it is running in the FPGAs. As mentioned in chapters 5 and 6, it is good practice to provide physical access to as many FPGA pins as possible so that they can be probed with normal bench instruments such as scopes and logic analyzers. Test points or connectors at key pins will hopefully have been provided by the board's designers for checking key

signals, resets, clocks, etc. However, most FPGA pins will usually be unreachable, being obscured as they are by the FPGAs ball grid array (BGA) package and therefore we need some other form of instrumentation.

The simplest type of instrumentation is to sample the pins of the FPGA using their built-in boundary scan chains, often referred to by the name of the original industry body that developed boundary scan techniques, JTAG (Joint Test Action Group). FPGAs have included JTAG chains for many years, primarily to assist test during manufacture and ensuring sound connection for BGAs. JTAG is most commonly known to FPGA users as one of the means for configuring the FPGA devices via a download cable. JTAG allows the FPGA's built-in scan chains to be used to drive values from the FPGA pins internally to the device and externally to the surrounding board. Through intelligent use of the JTAG chains and intelligent triggering of the sample by visible on-board events, a JTAG chain can capture a series of static snapshots of the pin values, helping to add some visibility to a debug process.

Greater visibility at FPGA boundaries or at critical internal nodes is provided by instrumentation tools such as ChipScope and Identify as described in chapter 3. As with any of these tools, there is an inverse correlation between how selective we are in our sampling 'vs' the amount of sample data. Since data is usually kept in any unused RAM resources available in the FPGA, we should expect that we will not be able to capture more than a few thousand samples of a few thousand nodes in the FPGA. Therefore we need to use some of our own intelligence and debug skills in order focus the instrumentation on the most likely sites of the fault and its causes.

A good Design-for-Prototyping technique is for the RTL writers to create a list of the key nodes in their part of the design i.e., a "where would you look first" list. This list would be a useful starting point for applying our default instrumentation.

11.6.3. Logic debug visibility

There is a traditional perception that FPGA-based prototyping has low productivity as a verification environment because it is hard to see what is happening on the board. Furthermore, the perception has been that even when we can access the correct signals, it is difficult to relate that back to the source design.

It is certainly true that FPGA-based prototypes have far lower visibility than a pure RTL simulator, however, that may be the wrong comparison. The prototype is acting in place of the final silicon and as such it actually offers far greater visibility into circuit behavior than can be provided from a test chip or the silicon itself. Furthermore the focus of any visibility enhancement circuits can be changed, sometimes dynamically.

As a short recap on the debug tools explored in chapter 3, we can gain visibility into the prototype in a number of ways; by extracting internal signals in real-time and also by collecting samples for later extraction and analysis.

Real-time signal probing: in this simplest method of probing designs' internal nodes, we directly modify the design in order to bring internal nodes to FPGA pins for real-time probing on bench instruments such as logic analyzers or oscilloscopes. This is a common debugging practice and offers the benefit that, in addition to viewing signals' states, it is also easier to link signal behavior with other real-time events in the system.

Embedded trace extraction: This approach generally requires EDA tool support to add instrumentation logic and RAM in order to sample internal nodes and store them temporarily for later extraction and analysis. This method consumes little or no logic resource, very little routing resource and only those pins that are used to probe the signals.

We shall also look at two other ways of expanding upon debug capability, especially for software:

Bus-based instrumentation: some teams instantiate instrumentation elements into the design in order to read back values from certain key areas of the design, read or load memory contents or even to drive test stimulus and read back results. These kinds of approaches are often in-house proprietary standards but can also be built upon existing commercial tools.

Custom debuggers: parts of the design are “observed” by extra elements which are added into the design expressly for that purpose. For example, a MicroBlaze™ embedded CPU is connected onto the SoC internal bus in order to detect certain combinations of data or error conditions. These are almost always user-generated and very application-specific.

11.6.4. Bus-based design access and instrumentation

The most commonly requested enhancements to standard FPGA-based prototyping platforms are to increase user visibility and access to the system, including remote access. We have seen how tools like Identify® and Xilinx® ChipScope tools offer good visibility into the prototype but these communicate with their PC-hosted control and analysis programs via the FPGA's JTAG port. As mentioned, the bandwidth of the JTAG channel can limit the maximum rate that information can be passed into or out of the prototype. If we had a very high bandwidth channel into the prototype, what extra debug capability would that give us?

Debug situations where higher bandwidth would be useful include:

- High speed FPGA configuration
- High performance memory access
- High speed download and upload of software images
- On-the-fly capture and pre-setting of memories
- High-performance data streaming
- Remote configuration and management

We can provide a higher bandwidth interface by using a faster serial connection or by using a parallel interface, or even a combination of the two. Very fast serial interfaces from a host directly into FPGA pins on a board is a difficult proposition but we could envisage a USB 2.0 connection and embedded USB IP in the design which might be used for faster communication, replacing the standard JTAG interface.

A simpler approach used by a number of labs is the instantiation in the design of blocks which can receive data in a fast channel and distribute it directly into parts of the design via dedicated fast buses. The instantiated block could be a simple register bank into which values can be written which then override existing signals in the design. Another use of a bus-based debug block might be to act as an extra port into important RAMs in the design, for example, changing a single-port RAM into a dual-port RAM so that the extra port can be used to pre-load the RAM.

The advantages of this kind of approach are clear, but even with the use of our chosen hardware description language's cross-module references this might mean some changes to the RTL. However, much of this change could be automated, or inserted after synthesis by a netlist editor. It might even be adopted as a company-wide default standard, much as certain test or debug parts are added into SoC design for other purposes during silicon fabrication.

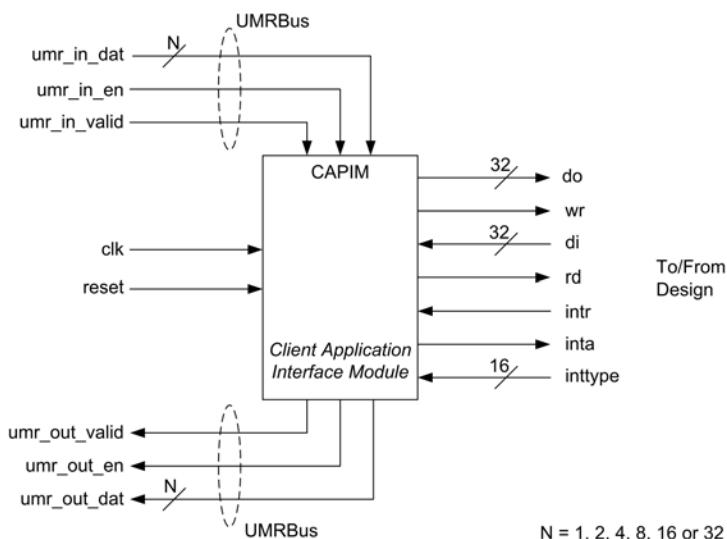
These advanced communication and debug ports are traditionally the reserve of tools which work on emulator systems but are starting to become more common in FPGA-based prototypes as well. One example of this is the Universal Multi Resource bus (UMRBus[®]) interface originally developed by one of the authors of this book, René Richter, along with his development director at Synopsys, Heiko Mauersberger (in fact, their names were the original meaning of the M and the R of UMRBus).

UMRBus, as the new name suggests, is a multi-purpose channel for high-bandwidth commutation with the prototype. It works by placing extra blocks into the design and linking them together via a bus-based protocol which also communicates back to PC-host. There are a number of blocks and other details which we will not cover

here, but at the heart of the UMRBus is a simple block called a client application interface module, or CAPIM, a schematic of which is shown in Figure 140.

As we can see in the diagram, a CAPIM appears as a node on a ring communication channel, the UMRBus itself, which carries up to 32 bits of traffic at a time. We can choose a width which best suits the amount of traffic we want to pass. For example, for fast upload of a many megabytes of software image, we might use a wider bus but for setting and reading status registers in the design a 4-bit bus might suffice, saving FPGA resources. Each CAPIM is connected into the design, often using XMRs to save boundary changes, to any point of interest.

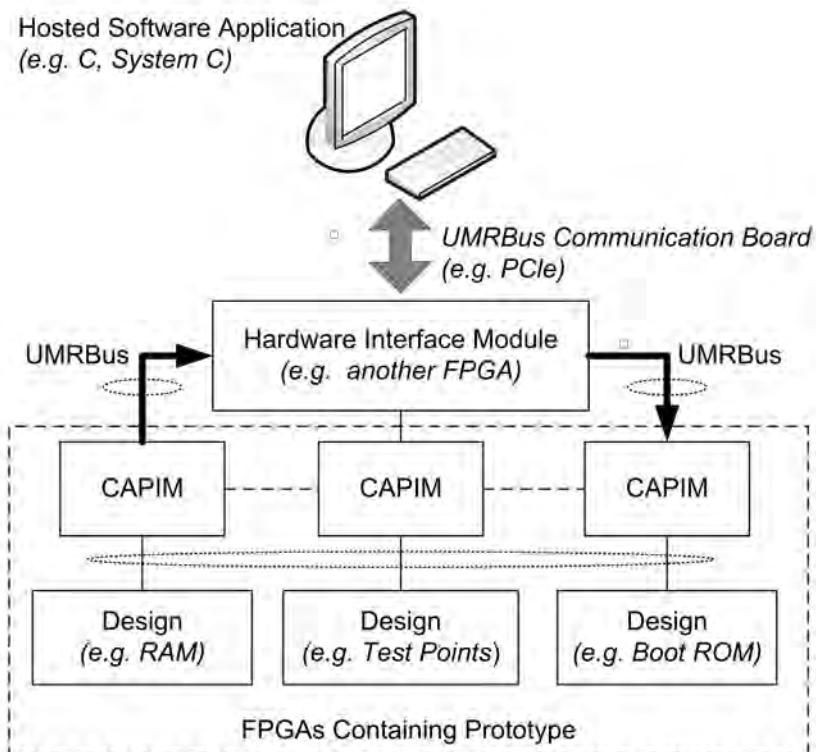
Figure 140 : Client application interface module (CAPIM) for UMRBus®



In Figure 141, we can see the use of three CAPIMs on the same UMRBus, each offering access and control of a different part of the prototype. In this example, UMRBus is allowing read and load of a RAM, of some simple registers and test points or to allow reprogramming of the book code in the design.

These functions might all be in the same FPGA or spread across the board across different segments of a UMRBus. Similar in-house proprietary bus-based access systems should also be designed to allow for multi-chip access and cross-triggering.

Figure 141 : UMRBus with three CAPIMs connected into various design blocks



11.6.5. Benefits of a bus-based access system

Some labs have developed their own variations on this bus-based approach and each will have its own details of operation, however, the general aims and benefits are the same as those listed above, so let's explore how this extra capability can help the FPGA-based prototyping project.

- **High speed FPGA configuration:** higher bandwidth access to the prototype significantly speeds up the speed at which designs can be downloaded onto the board compared to serial methods typically used. This is especially valuable early in the design cycle when hardware-related design changes most often occur and we are debugging the design's first

runs on the board. The use of a GUI or a command-line interface for configuration and programming benefits greatly from “instant” access and fast configuration, avoiding those irritating delays for a few minutes configurations time.

- An example of a configuration approach which uses a proprietary bus-based interface is the CONFPRO unit from Synopsys which uses the UMRBus protocol to send large amounts of data over USB to an embedded supervisor on the FPGA board which configures the FPGAs in one of their faster parallel modes (see chapter 3) rather than via a JTAG cable.
- **High performance memory access:** the ability to read and write directly to the memory on the FPGA-based prototype can dramatically reduce bring-up time. On a prototype there can be many megabytes of the FPGA’s internal RAM in use at any time. Memory pre-load and read-back functions can be more easily implemented if an extra bus is placed into the prototype for that purpose rather than trying to employ the SoC’s own CPUs and buses to achieve the same result. We can avoid software rework or scheduling issues involved in having the SoC CPUs simply listen to a host-controlled port and pass data to a RAM.
- Direct access to memory via something like the UMRBus enables us to view memory contents and also rapidly download, upload and compare large quantities of memory content under script control or via TCL commands. A lab library of pre-defined interface objects might be available for connection to our debug bus, such as memory wrappers, which provide a second port into a RAM. This minimizes the need for on-the-spot modeling of many components and speeds access to the system during debug. For example, Synopsys keeps pre-defined IP in the form of a UMRBus-to-SDRAM component, which enables direct access to SDRAM for programming, pre-load and read-back without re-synthesis and/or place & route changes.
- **High speed upload of software images:** a specific use of the fast-memory access is for loading software images. Since the major use of the prototype may be for enabling a fast and direct platform for the software team, we should enable their normal fast and iterative working methodology. A software image ready for loading into the CPU might be very quickly generated using the normal compile and linking tools. It would then be irritating if it took far longer to load the result into the platform in order to run in. Estimates by colleagues using JTAG-based interfaces tell of 30 minutes to upload a typical software image. This could be cut to considerably less than a minute using a higher-bandwidth interface.
- **High-performance data streaming:** another use of the faster access into the prototype might be to input data streams from the host at a rate fast enough to fool the SoC design into thinking that it is coming from the real

world. Some types of data input might not be suitable for this approach, for example, interactive data or data with a high-rate of delivery such as raw network traffic. Other data, such as imaging or audio data would fit particularly well into this approach. We could, for example, deliver recorded video to the SoC from the host via a bus-based interface to test the software's capability in a certain video processing task.

- It is a short step from this proprietary data delivery to adopting an industry standard for passing data and even transactions into and out of the prototype. We shall explore this area further in chapter 13.
- **Remote configuration and management:** with advanced bus-based access, we can remotely access and program the prototype via a host workstation. Board types and configurations could be scanned and their setup requirements automatically detected. Board initialization, configuration and monitoring could then be handled remotely, lightening the burden for a non-expert end-user. For this, it may be necessary to drive the bus-based access from a standard interface port, such as USB. This could be performed via a hardware adaptor which runs the bridge from PCIe (in this case) to the on-board bus access. In the HAPS-60 series of boards, a hardware interface module called CONFPRO connects the PCIe interface on the host workstation to the UMRBus interface on the prototyping system.

In very advanced cases, the bus-based access can become a transport medium for protocol layers but this might be a large investment for most project-based labs, therefore we might expect these types of advanced use modes to be bought in as proven solutions from commercial board and tool vendors.

Once we start to employ the most sophisticated methods for debug and configuration of the system, we might begin to explore other use modes including:

- Direct link to RTL simulator
- Support for transaction-based interface via SCE-MI
- Hybrid system prototyping with virtual platforms

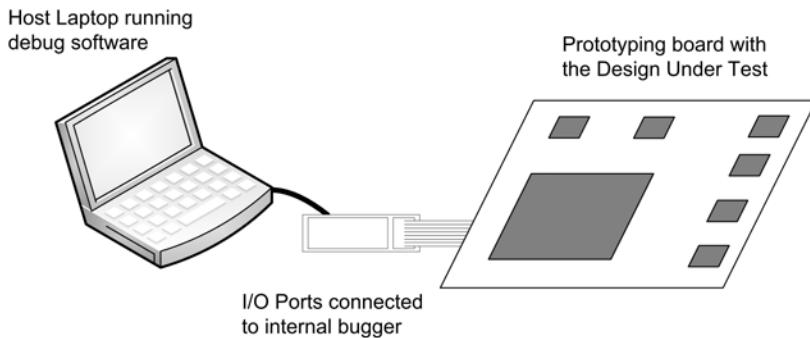
We shall explore these in chapter 13.

11.6.6. Custom debug using an embedded CPU

Most of the SoC designs which have embedded processors would also have built-in custom debuggers. These custom debuggers would be used to connect to the processor bus present in the SoC designs, usually to load software programs to be run on the processor, debug the software programs by single stepping or breakpoints and access all the memory space of the processor bus. These custom debuggers will

be present inside the design which will be connected to the external world through some dedicated IO ports. Usually a debugging software utility running on a laptop will be connected to these FPGA IO ports using some hardware connected through USB or other IO ports as shown in Figure 142. Using the software utility we can load programs into the program memory, debug the software and access the memory space in the SoC.

Figure 142: Software debugger linked into FPGA-based prototype



If the design to be prototyped has this debugger, then it can be used to debug the design if it doesn't work on the board. We should take the dedicated IO ports coming from the inbuilt debugger out of the prototyping board using IO boards or test points. Then the debugging software utility can be connected to these IO ports. After programming the FPGAs, the software utility can be asked to connect to the internal debugger present in the design. Upon successful connection to the internal debugger, we can try to read and write to all the different memory spaces in the processor bus. Then a simple program can be loaded and run on the processor present in the design. Finally the actual software debugging can happen over the real design running on the prototyping board.

11.7. Note: use different techniques during debug

Some might say that on-the-bench debug of any design, not just FPGA-based prototypes, is more intuition than invention. Debug skills certainly are hard to capture in a step-by-step methodology, however, it is a good start to put a range of tools in the hands of experienced engineers. On the bench, it is common to find a combination of these different debug tools in use, such as real-time probing, non real-time signal tracing and customer debuggers.

For example, if a certain peripheral like a TFT display driver in a mobile SoC chip is functioning improperly then we might try to debug the design though the custom debugger first. We could read and write a certain register embedded in the specific peripheral through the customer debugger to check basic access to the hardware. Then we can try to tap onto the internal processor bus through Identify instrumentor and debugger by setting the trigger point as the access to the address corresponding to the register. With the traced signals we can identify whether proper data is read and written into the register. Then we can connect the oscilloscope or logic analyzer on the signals which are connected to the TFT display to check whether the signal are coming appropriately when a certain register is read and written. Of course, all of these steps might be performed in a different order as long as we are successful. With this combined debug approach we can debug most of the problems in the design.

11.8. Other general tips for debug

One of the aims of the FPMM web forum which accompanies this book is to share questions and answers on the subject. The authors fully expect some traffic in the area of bug hunting in the lab and to get us started, here is a miscellaneous list, in no particular order, of some short but typical bug scenarios and their resolutions.

Many thanks to those who have already contributed to this list.

- **Divide and conquer:** if there are multiple interfaces to the external chipsets, we can test them one by one. Start with the obvious and visible before moving to the obscured and invisible. For example, taking one non-working interface at a time, the interface signals can be probed to see whether the protocol is followed as per the expectations. If they were not followed, we can probe the internal part of the design flow (may be a state machine) which produces those external signals inside the FPGA using Identify or ChipScope, to check whether the flow is proper inside the design. By this way we can test and make sure all the individual interfaces are operating as expected.
- **Check external chipsets:** the problem may not be in the FPGAs at all. If an external chipset is showing unexpected behavior, then we can try to test the design on the board using synthesizable model of such chipsets implemented temporarily in an FPGA instead. If the synthesizable model is not available for the chipsets then we can consider creating such models themselves or use a close equivalent available as open source. If the functionality is very complex, then at least a model which takes care of the interface part giving some dummy data would be sufficient to test the main design. The open source website www.opencores.org is a good source for synthesizable RTL. Some chipsets have data such as vendor ID or release

version hardcoded within them, which can be read through the standard interface. The testing of those external chipsets can be started by reading and verifying such hardcoded data first. This way we can make sure that the interface between the external chipset and the design on the prototyping board works correctly.

- **Tweak IO delays:** if the input and output delay requirements are not met then the sampling of data inside the FPGA and external chipset will be affected, which can cause data corruption. If this is suspected, then we can try to introduce IODELAY elements present in Xilinx® FPGAs in the data and clock path. IODELAYs are programmable absolute delay elements which can be applied in primary IO paths inside FPGAs. By varying the delay in these blocks, we can make sure that the inputs and outputs are sampled correctly.
- **Tweak clock rate:** clock rate can be suspected when the design on board is not dead but the operation of the design is not as expected. For example, reading and writing to a memory space might be working but the data read might not be consistent with the written data. If timing issues are suspected (and the PLLs etc. allow running at reduced rates) then we may try to run the system temporarily at reduced clock rate. Often the system can show more signs of life at the reduced clock.
- **Understand critical set-up needs:** for the design to behave properly, internal blocks might have to be configured in a certain way but some are more critical than others. We should like to be able to rely on documentation sent from the SoC team which highlights the most critical setup. For example, there could be an internal register controlling the software reset of the entire design which might have to be written with a valid value initially to make the design work. As a further example, there could be a mode register which, by default, could be in sleep mode that needs to be written with a valid value to make it active. Similarly some of the external chips might have to be configured in a certain way to make the whole prototyping system work. For example, to make a camera image sensor to send proper images to the FPGA, the internal gain registers in the sensor might have to be configured to a certain non-default value.
- **Check built-in security:** documentation should record any necessary security or lock cells in the design or external IP. For example, a security code which checks for certain values in an internal ID register, but omitting it means that the prototype will seem alive but unresponsive. As another example, the boot code running the internal process might expect a key bitstream to be present in a flash memory or an external ROM in order to proceed. In such scenarios, it is worth asking again if any such keys are required in the SoC design, just in case documentation has not kept up with the RTL.

- **Challenge basic assumptions:** one obstacle to debug can be an assumption that the obvious is already correct. We are not asking to check if the power cable is connected (there is a limit) but some basic assumptions should be re-checked, such as little endian-ness and big endian-ness between the buses on FPGAs and any external chipsets or logic analyzers added for prototyping purposes. Another example is to check if any transmit and receive ports in the design should be connected to the external chipsets directly or with crossed wires.
- **Terminations:** many interface formats such as I2C or PCI or SPI, etc. need the physical lines on the board to be terminated in a certain way. We could check if those lines are terminated as required, if connectors are intermittent or if there are local voltage rails issues preventing correct termination.
- **Check IP from the outside-in:** while testing the IPs, test the interfaces to the IPs and not the functionality of the IPs themselves. After all, the IPs should have been tested exhaustively by the IP vendor themselves so at least we can initially assume that the IP should work as expected on the FPGA. However, if the IP has been delivered as RTL and not supported for use on FPGA by the vendor/author then we might need to challenge this assumption. IP interfaces can be probed out through logic analyzers or embedded debuggers like Identify or ChipScope.
- **Reusing boards allows shortcuts:** we could start a debug process from the IPs or design blocks which were known to be working well on the board in a previous project. For example, if the design has a processor and many peripherals we might start with a USB IP which was successfully prototyped earlier. By making sure that the USB IP works fine along with the processor, we can get assurance that at least some software code and the bus through which processor and peripherals communicate is also functioning properly. Then we can debug the new peripherals from a good foundation.
- **Test GTX pins first, test protocol second:** if the design uses any high speed gigabit transceivers then we can use the integrated bit error ratio tester (IBERT) built into the Xilinx® ChipScope tools. IBERT helps to evaluate and monitor the health of the transceivers prior to testing the real communications through that channel on the board. Only when we are sure about the physical layer should we move on to use our protocol analyzers or bus-traffic monitors.
- **Use sophisticated analyzers:** these analyzers can offer real-time protocol checks, bus performance statistics and can monitor bus latencies. We can also add external bus exercisers to more easily force behavior on the system buses so interference or other de-rating is taking place on the bus inside the prototype. If external equipment is not available, then custom-

written synthesizable bus exercisers and bus analyzers might be used. We could consider adding a known, simple checker into the design in order to later use it for checking good bus behavior or forcing traffic.

These are a few of the ideas that Synopsys support staff and end-users have shared over many years of successful prototyping and at time of writing, the authors look forward to hearing many more as the FPMM website goes live.

11.9. Quick turn-around after fixing bugs

Debugging can become an iterative process and given that the time taken to process a large SoC design through synthesis and place & route might be many hours, if we find ourselves wasting a lot of time just waiting for the latest build in order to test a fix, then we are probably doing something wrong.

Here are three approaches we recommended in order to avoid this kind of painful “debug-by-iteration.”

- Stick to a debug plan
- Use incremental tool flows
- Both of the above

A debug plan for a prototype is much like a verification plan for the SoC design as a whole. As a full rerun of the prototype tool chains for large designs might take more than a whole day, we need to have clear objectives for the build, which we should aim to meet before re-running the next build. A debug plan sets out the parts of the design which will be exercised for each build and also a schedule of forthcoming builds. This is particularly useful when multiple copies of the prototype are created and used in parallel. Revision control for each build and documentation of included bug fixes and other changes are also critical. This is all good engineering advice, of course, but a little discipline when chasing bugs, especially when time is short, can sometimes be a rare commodity. With a good debug plan in place and a firm understanding of the aims and content of each build, a regime of daily builds can be very productive.

A day’s turn-around is an extreme example and in some cases it would be much less than that. For example, a bug fix that requires a change to only one FPGA will not require the entire flow to be rerun. The partitioner may work on the design top-down, but for small changes, the previous partition scripts and commands will still be valid. As mentioned in chapter 4, a pre-synthesis partitioning approach helps to decrease runtime in this case because synthesis and place & route take place only on the FPGA that holds the bug fix.

To achieve a faster turn-around on small changes we can use incremental flows and we shall take a closer look at these now.

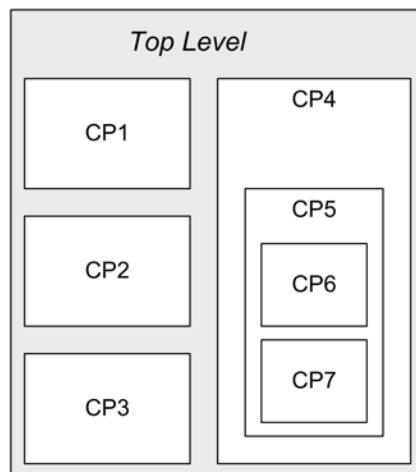
11.9.1. Incremental synthesis flow

As mentioned in chapter 3, both synthesis and place & route tools have the ability to reuse previous results and only process new or changed parts of the design. Considering synthesis first, the incremental flow in most tools was created to reduce the overall runtime taken to re-synthesize a design when only a small portion of the design is modified. The synthesis will compare parts of the design to the same parts during the previous run. If they are identical then the tools will simply read in the previous results rather than recreate them. The tool then maps any remaining parts of the design into the FPGA elements instead of the entire design, combining the results with those saved from the previous runs of the unchanged parts in order to complete the whole FPGA. This incremental approach can save a great deal of time and takes relatively small effort to set up.

In Synopsys FPGA tools, incremental synthesis is supported by the compile point synthesis flow. In the compile point flow, we manually divide the design into a number of smaller sub-designs or compile points (CPs) that can be processed separately. This does not require any RTL changes but is controlled by small changes in the project and constraint files only, and can even be driven from a GUI.

A CP covers the subtree of the design from that point down although CPs can also be nested as we can see in Figure 143.

Figure 143: possible arrangement for Compile Points during FPGA Synthesis



The design can have any number of compile points, and we do not have to place everything into CPs because the tool makes the top-level a CP by default. Therefore the simplest approach might be to define CPs on all the modules which are frozen or are not expected to change, allowing the tools to simply reuse the previous results for that CP.

Since each CP will be synthesized separately, we must define a separate constraint file for each one which will include many of the same clock and boundary constraints that we would need if the CP block was being assembled in a bottom-up flow, or as a standalone FPGA. Details are of constraints available in the references but the aim is that for a small up-front effort in time budgeting or by limiting our CP boundaries to registered signals, we can dramatically reduce our turn-around time.

The tool needs to be able to spot when the contents of a CP have changed it achieves this by noticing when the CP's RTL source code logic or its constraints have changed. Changes to some implementation options (such as retiming) will also trigger all CPs to be re-synthesized.

There is no such thing as a free lunch, however, and in the case of incremental flows the downside is that there may be a negative impact on overall device performance and resource usage. This is because cross-boundary optimizations may be arrested at CP boundaries and so some long paths may not be as efficiently mapped as they would have been had the whole design been synthesized top-down. In Synopsys FPGA synthesis this can be mitigated to some degree and the amount of boundary optimizations across the CP boundary can be controlled by setting the CP's type as either "soft," "hard" or "locked."

Another reason that incremental flows might not yield the highest performance results is that the individual timing constraints for each CP may not be accurate as they rely on estimated timing budgeting or user intervention. Conversely, this might be seen as an advantage since we might purposely choose to tighten or relax the constraints for each CP in order to focus the tool's effort on more difficult-to-achieve results for certain parts of the design, while relaxing others to save area or runtime.

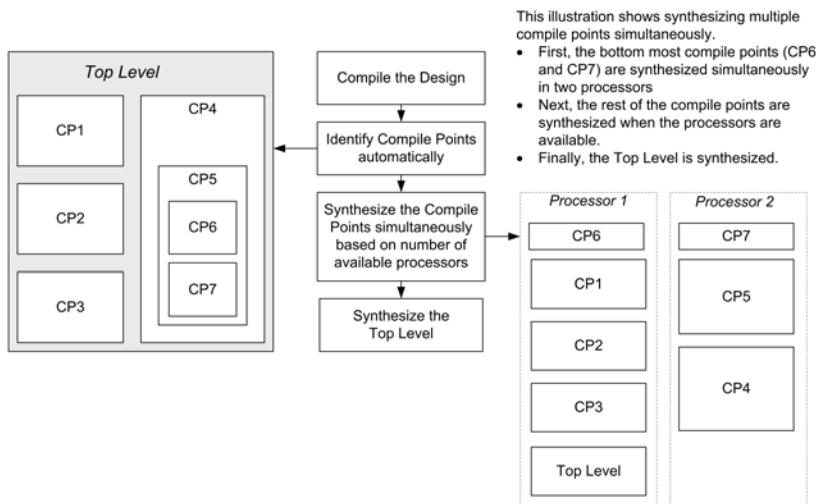
Readers might be struck by the similarity between CP incremental synthesis and traditional bottom-up synthesis of a design block-by-block. However, the scripting, partitioning and resource management involved in a traditional bottom-up flow is sometimes seen as too much of an investment for a prototyping team. There is also the problem of tracking exactly which files are to be re-synthesized in the new build. In a design of thousands of RTL files, automation is very desirable.

11.9.2. Automation and parallel synthesis

If the workstation upon which the FPGA synthesis is running has multiple processors then the synthesis task can be spread across the available processors, as shown in Figure 144.

With such multiprocessing enabled, multiple CPs can be synthesized simultaneously further reducing the overall runtime of synthesis even for the first run or complete

Figure 144: Automatic compile points and synthesis on multiple CPUs



re-builds when all CPs are re-synthesized. By using the automatic CPs and multiprocessing options together, we can reduce the overall runtime of the synthesis during the first time as well as the subsequent incremental iterations.

A typical SoC design would need a number of CPs to be defined with individual constraint files in order to achieve a significant reduction in the synthesis runtime. Again, if the up-front investment for an incremental flow is too great or requires too much maintenance (e.g., in scripts) then the flow becomes less attractive. Indeed, defining a large number of CPs and creating constraints files for each may even be seen as too time consuming. To overcome this apparent hurdle, Synopsys FPGA synthesis is able to create and use CPs automatically.

When automatic CPs are used, the tool can analyze a design and identify modules that can be defined as CPs. The CP's timing constraints can also be automatically budgeted from the top-level constraints. This eliminates the need for us to define a separate constraint file for each defined CP.

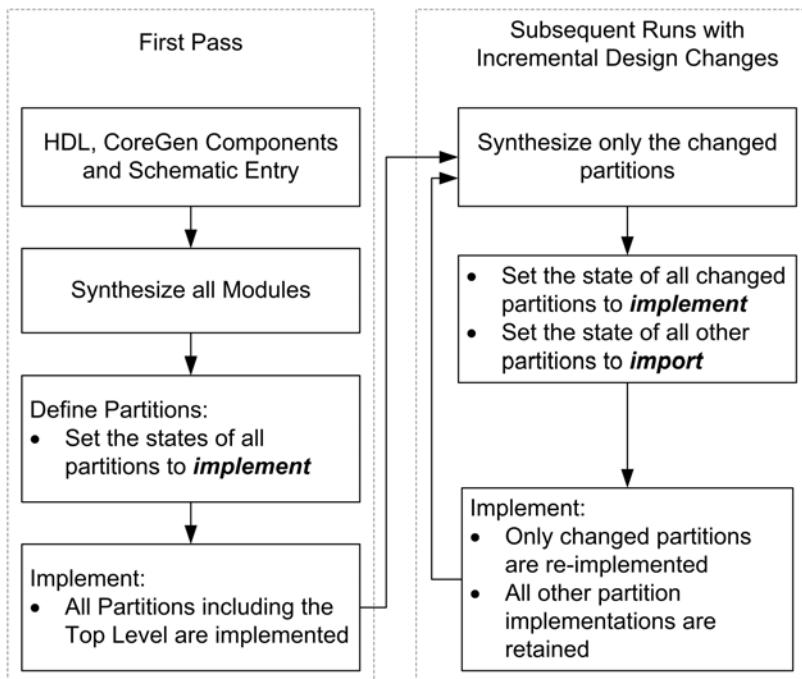
This is a relatively new technology and is a useful weapon for trading off runtime against overall performance.

11.9.3. Incremental place & route flow

Incremental flow in place & route can be achieved using design preservation flow in the Xilinx® back-end tool, the flow for which is seen in Figure 145.

Design preservation is one of the hierarchical design flows supported in the Xilinx® back-end tool. In the design preservation flow, the designs are broken into blocks referred to as partitions. Partitions are the building blocks of all the hierarchical design flows supported in the Xilinx® back-end tool. Partitions create boundaries

Figure 145: Design preservation flow in Xilinx® ISE® tools



around the hierarchical module instances so that they are isolated from other parts of the design.

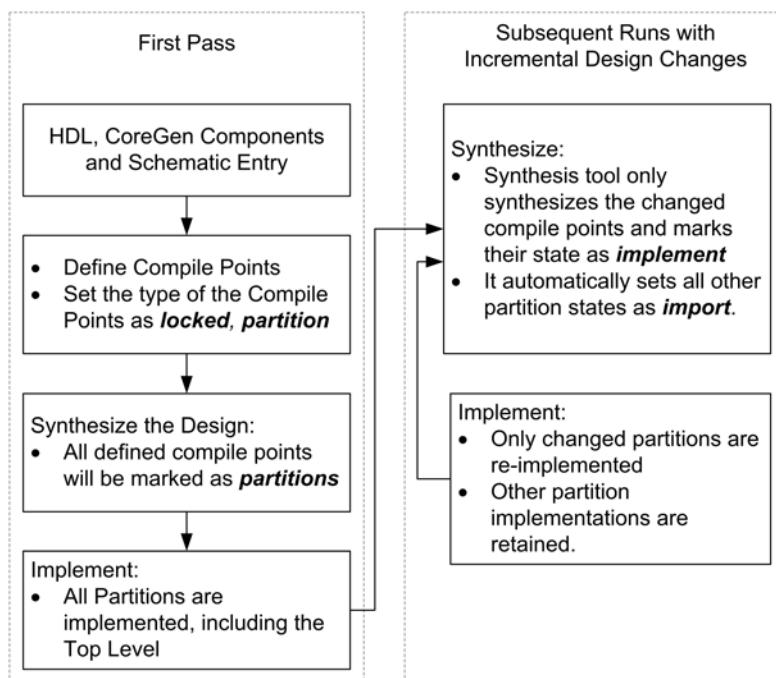
A partition can either be implemented (mapped, placed and routed) or its previous preserved implementation can be retained, depending on the current state of the

partition. If a partition's state is set as "implement," then the partition will be implemented by force. And if the partition's state is set as "import," then the implementation of that partition from the previous preserved implementation will be retained.

When a design with multiple partitions is implemented in the Xilinx® back-end tool for the first time, the state of all the partitions must be set to "implement." For the subsequent iterations, the state of the partition should be set either to "import" if the partition has not changed or "implement" if it has changed. This way, only the changed modules are re-implemented and the implementations of all the partitions that have not changed are retained. This saves significant time in the implementation of the entire design.

11.9.4. Combined incremental synthesis and P&R flow

Figure 146: Combined synthesis and place & route incremental flow



By combining incremental flows for synthesis and for place & route we can gain our maximum runtime reduction. In fact, since place & route runtime is typically

much longer than synthesis runtime, setting up incremental synthesis without incremental place & route would not really improve turn-around time that much.

With this in mind, Synopsys FPGA synthesis has been created to interleave its compile point flow automatically with the Xilinx® design preservation flow, as shown in Figure 146. In this integrated flow, the type of CP in the synthesis flow should be set to “locked, partition,” so that it will be marked as a partition during the place & route stage. When the integrated flow is enabled, the Synplify® Premier software automatically writes the states of the partitions as “implement” or “import” as appropriate.

By using this integrated flow, as summarized in Figure 146, the overall turnaround time to synthesis, place & route and design can be significantly reduced.

11.10. A bring-up and debug checklist

Table 30: A step-by-step approach to bring-up and debug of the prototype in the lab

1	Check all power rails	<i>Are voltages stable, connections good, current limits inactive?</i>
2	Check all input clocks	<i>Frequency, quality and correct phase relationships</i>
3	Check FPGA configuration and reset	<i>Do FPGAs come out of reset on the same clock edge?</i>
4	Load small RTL design	<i>Checks all the above, confirms board hardware ready</i>
5	Add instrumentation	<i>Get familiar with working debugger using small RTL design</i>
6	Load small self-running program into internal CPU	<i>Test write and read of key registers and internal RAM.</i>
7	Test external RAM	<i>Interface to RAMs may have been made especially. Debug separately.</i>
8	Load “Hello World” program into internal CPU	<i>Tests CPU-to-host comms. Test comms to external IP.</i>
9	Load whole SoC prototype	<i>Including internal IP and memories</i>
10	Load real software image	<i>Prototype is now useful to end-users; open the Champagne!</i>

We have covered a lot of ground in this chapter but there will be very many on-the-

bench debug scenarios not covered here. Every design and lab setup is different and we can only attempt to cover the more common patterns in the projects that we know.

We hope that there will be a great deal of discussion and “*I found this, try that...*” experience swapped between prototypers on the FPMM online forum, from which we can all learn to become better debuggers. In the meantime, Table 30 gives a rough checklist of tasks in the lab during bring-up and debug.

11.11. Summary

After giving ourselves confidence that the FPGA board is fully functional and configured correctly and also having checked the implementation and timing reports for common errors, we can consider that functional faults in the operation of the prototype might actually be bugs in the design itself that are becoming visible for the first time.

As we find and correct bugs and move towards a fully functional prototype we are being paid back with interest for all our hard work in getting this far. The prototype has also become an extraordinarily powerful tool for validating the integration of the software with the SoC. Replicating the prototype for use by a wider community of software engineers is relatively quick and simple at this stage.

The authors gratefully acknowledge significant contribution to this chapter from

Ramanan Sanjeevi Krishnan of Synopsys Bangalore

There are situations in which taking a prototyping system outside the development lab can benefit the development and the overall success of the product. Having a portable prototyping system can provide developers with the ability to interact with other parts of the system not available in the lab, and showcase the product before silicon is available. This chapter describes the issues and effort that it takes to make a hardware prototyping system portable, so it can be used outside the R&D lab as well .

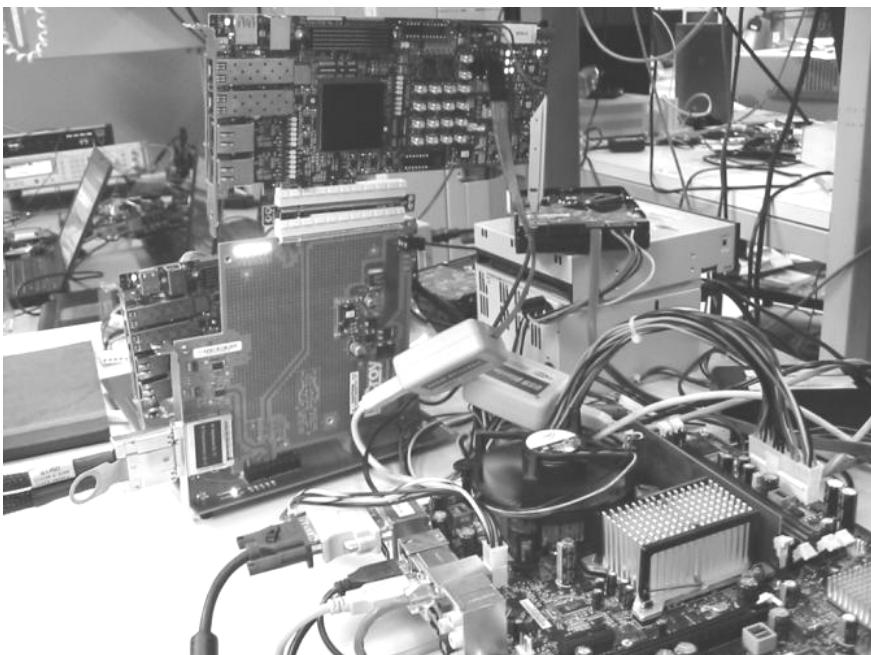
12.1. The uses and benefits of a portable prototype

The hardware prototyping system is usually considered very close the final SoC as far as its functionality and performance, however, its form-factor is often so far from the final SoC package that it isn't even funny. Try as we might to keep things tidy and regular, a typical on-the-bench prototype often looks like the result of barely controlled chaos . . . but it only *looks* that way.

As we assemble our rig on the bench, such as the typical one shown in Figure 147, we of course know exactly why each piece of equipment is present and how it contributes to the overall success of the project. In fact, the project shown in the photo is a Synopsys® lab test of our latest PCIe Gen3 IP design, which at time of writing was proceeding very well and providing unique feedback on the performance of the IP as only an FPGA-based prototype can.

What if the prototype could be constructed small and robust enough so that we could take it to out of the lab? What extra benefits, which may prove critical to the overall success of the SoC project, could be derived?

Figure 147: A typical bench-bound prototype!



Before we proceed to explore how we might achieve portability, let us revisit those benefits of prototyping that we listed in chapter 2 in order to highlight the main reasons why we might want to take our prototype out of the lab.

- **Field tests:** our application may have functions which require field tests under real-world conditions that are not available in the lab. A good example is a mobile radio transceiver which needs to communicate with a base station under many and varied ambient conditions.
- **Local compliance tests:** wireless standards evolve at different speeds and may have variations for different geographies. For compliance we often require that air interface tests are performed in that geography.
- **Early customer demo:** a portable prototyping system at a potential customer site can show off the product's capabilities and readiness for market.
- **Partner co-development:** early pre-silicon access to the system prototype enables product development and test that spans multiple geographies, often involving a number of companies in partnership.
- **Exhibition demo:** being first to demonstrate a new functionality or new standard implementation can gain great momentum in a market..

- **Investor status check:** a portable prototyping system can be used to demonstrate a product's capabilities to a board of directors or potential investors alike. Our audience can quickly and effectively evaluate our progress when presented with a live demo of the final product's prototype.

Each of the above may be good reasons for any particular team creating a more portable prototype than the bench-bound example above. In fact, the Synopsys IP team responsible for that prototype did package it up for demonstration out of the lab. Torrey Lewis, IP developer at Synopsys, Hillsboro Oregon, explains why they did that extra work:

"It is definitely a benefit to be able to travel with the prototype. For our PCIe Gen3 setup, we've been able to travel to some conferences for display in the Synopsys booth. The PCIe Gen3 spec is not yet finalized, so even more interesting for me as an engineer has been the opportunity to do early testing on-site with industry leaders. This has been mutually beneficial for all concerned."

Let us now go on to see how we can make a prototype which is able to break its bonds to the bench.

12.2. Planning for portability

The best time to plan for portability is when we start our prototyping project. It is best to plan for prototyping system portability at the project start and make provisions for it as the system and the SoC is developed. For example, if we are aiming at a multi-board system comprising one main board with additional daughter boards then we must ensure that our arrangement can be made robust when necessary.

Javier Jimenez of DS2 also feels that reliability is important in their out-of-lab trials of their Broadband-over-Powerline prototypes, as he explains below:

"Some of our field trials are performed by engineers from different disciplines, for example, protocol experts or software engineers. These are not hardware experts and so they could not be expected to perform on-the-spot fixes to circumvent any board failures. Therefore the FPGA boards, and indeed the whole prototype must be of high quality and have high mechanical reliability."

You can read more about DS2's project in chapter 2. Meanwhile, the following sections explore the critical items for building a prototyping system that is portable.

12.2.1. Main board physical stiffness

The obvious first concern is that the board itself should have inherent stiffness and not flex in out-of-lab situations. The main board should be stiff enough as to not flex either lengthways or crossways. Since today's FPGAs are most useful for prototyping in ball-grid array packages, the strength and quality of the FPGAs solder-ball connections to the motherboards is critical to the overall system reliability. Little or no flexing of the board can be tolerated unless these ball connections are broken.

Typical prototyping boards with multiple high-pin count FPGAs will end up having over 20 layers and become fairly stiff on their own. However, it is still recommended to make space for stiffeners in both board's length and width so that they may be mounted when necessary. Alternatively, the board might be slipped into a frame which hold the board at its edges, however, this is not as mechanically sound as on-board mounted stiffeners.

12.2.2. Daughter board mounting

Daughter boards are added to the main board for a variety of reasons such as adding fixed hardware to the FPGA system. When these boards are designed to work with the main FPGA board, we must consider the mechanical, electrical and thermal aspect of the combined assembly as described in the following paragraphs:

- **Stacked configuration:** in this configuration, the daughter board is mounted over the main board and its footprint overlaps the main boards. The advantage of such configuration is the smaller overall footprint. The disadvantages are the possibility of restricted probing access on the main board and the risk of restricted air flow over the FPGAs. Both main and daughter boards must be designed for solid mounting of the two.
- **Lateral configuration:** in this configuration, the daughter board is mounted next to the main board. This is typically the case when the daughter board is not designed in house, rather purchased or provided by a third-party IP vendor and its mechanical configuration is unrelated to the main board. The advantage of such configuration is having unrestricted access on the main board. The disadvantage is the greater overall footprint and tendency to be less robust.

12.2.3. Board mounting holes

Main board should have sufficient mounting holes to allow a secure and solid mounting to a chassis. It is recommended provide for mounting holes no more than

10cm apart from each other in either direction. Special attention should be paid to areas where the board is subjected to connector insertions. It is recommended to place mounting holes such that the mounting studs will absorb most of the connector's insertion and extraction forces that are typically perpendicular to the board. If additional daughter boards are to be added on top the main board, there should be allowance for mounting the daughter boards to the main board. This is discussed in more detail in the section below.

12.2.4. Main board connectors

Connectors are often a common point of failure in the prototyping system's reliability due to wear and improper mount and dismount cycles in a typical lab environment. Addressing the connectors' issues in advance can minimize this risk especially when we need to move the combined system. Connectors are typically made to mate with either in only one axis, so twisting or skewed removal or insertion may cause damage. We recommend adding adequate strain relief and retention in the insertion-removal axis; in fact, many connector schemes have an option for such retention clips already and it is a false economy not to install these by default.

Cables that exit the system's enclosure must be retained to the system's chassis rather than rely on the retention by the connector itself, which adds further strain onto the connectors and cables alike.

12.2.5. Enclosure

The enclosure is where all the pieces fit together for convenient transportation but at the same time some level of access and visibility is needed. When selecting an enclosure, the following items are typically considered:

- **Mechanical configuration:** even if the system is intended to be placed on the bench, many of the shelf enclosures are available in industry standard sizes so they can also be mounted into a standard rack. The additional rack mounting hardware can easily be removed if found to be undesired.
- **Main board mounting:** it should be mounted with as many screws/spaces as available.
- **Daughter boards:** as mentioned above, should be secured either to the main board (if stacked) or to the chassis with no mechanical stress between the daughter board and main board.

- **Power supply:** should be placed away from the main board to allow good clearance for main board cabling and cooling for the power supply. If the system is expected to travel to countries where AC is different than the country of origin, the power supply must be multi-voltage. If power supply has its own exhaust fan, it's important to mount it to the chassis in a way that does not obstruct the cold air inflow to the power supply module.
- **Internal cables:** should be retained on the boards they mount to.
- **External cables:** should also be retained to the chassis.
- **Access:** Most access to the system is done via the system's interface cables. However, some provision may be needed for debugging, so critical signals may be brought to debug connectors for added visibility. These debug connectors should be mounted on the chassis wall so opening the chassis will be not necessary.
- Miscellaneous items such as FPGA download module, or debugger modules should also be retained to the chassis.

12.2.6. Cooling

When in the bench, the FPGAs and other components are generally exposed to ambient airflow and may not require extra cooling. However, even on the bench, it is recommended that each FPGA should have its own heat sink and fan arrangement. This can be linked to a temperature system monitor in a feedback loop to ensure that the fans are only used when necessary (see chapter 5).

Even if on the bench we do not employ any cooling, when the prototype is encased and isolated from ambient air movements, the temperature can rise rapidly. It is recommended to equip the enclosure with cooling fans that will draw hot air from the enclosure and blow it to the outside. The key to effective cooling is the positioning the enclosure fan and cold air intake such that cold air is drawn from the intake holes and forced over the hot spots and out the enclosure via the enclosure fan. This is usually accomplished when the exhaust fan and the cold air intake holes are on opposite sides of the enclosure. It's recommended to block any openings that may be next to the enclosure fan so as to not "short circuit" the air flow over the "hot spots." It is still necessary to provide local heat-sinks and fans for the system's hot spots, such as FPGAs or power supply circuits.

12.2.7. Look and feel

For some out-of-lab uses, it is important that the prototype looks less like a school science project and more like a finished product. This can be achieved by, in effect, hiding away much of the hardware inside a professional looking case. This has the added bonus of making the prototype more tamper-proof and probably more reliable as a result.

Some commercial FPGA platforms are designed to be more robust and portable

Figure 148: CHIPit® Iridium, an encased prototyping system



than a typical open board. Examples of this type of system are the CHIPit® Platinum and CHIPit Iridium platforms, both of which have sophisticated FPGA boards encased in robust enclosures. The Iridium, shown in Figure 148, is a particularly interesting example because it is clearly still a prototyping platform, you can see the boards and chips in place, but by replacing the sides on the enclosure, the unit becomes much more like a finished product. This simple trick may count for a lot in a demonstration scenario.

The best places to go and see how teams overcome the portability challenges are so-called “plug-fest” events. These are organized by special interest groups, often representing a new industry interconnection or graphics standards. The aim is to bring together those teams which have hardware for driving the new standard so that they can physically plug their equipment together. Of course, all teams would have already been working to the standard specifications but there is nothing like the confidence given when the pre-silicon prototype actually works with another team’s prototype. Even if it doesn’t work straight away, this is a much better place to find out than after the product is released.

Critical to the success of plug-fests is the reliability of the equipment and even though much of it is in FPGA-based prototype form, we will see platforms carried between table-top lab set-ups or moved around on trolleys in order to perform as many experiments as possible. Turning up at a plug-fest with a “rats nest” of unreliable boards would not be making best use of the opportunity.

12.2.8. Summary

This has been a short chapter exploring the use of FPGA-based prototypes outside of the normal lab environment. It has hopefully given some more glimpses of the possibilities of prototyping as a whole, but the aim is really to ensure that we think ahead to create a more reliable and portable prototype. This will make it more tamper proof when used remotely by, say, software engineers and more likely that the prototype will survive the project and be useable for subsequent projects.

The out-of-lab use of prototypes can often be decided upon well into the project, for example at an exhibition opportunity that it just too good to miss, but the first silicon is still months away. Being ready for these kinds of events may be recognised by senior management as another reason to adopt FPGA-based prototyping in all SoC projects.

Having decided to invest in building an FPGA-based prototype, we should understand its full potential. This chapter shows how the prototype can become a part of a larger verification environment by linking it to simulators, emulators or virtual models in SystemC™. In other sections of this book we have focused on how to create FPGA prototypes. This chapter looks at how design teams benefit from integrating their prototypes into hybrid verification systems. We also look at the enabling technologies for hybrid verification solutions, including the verification interfaces themselves.

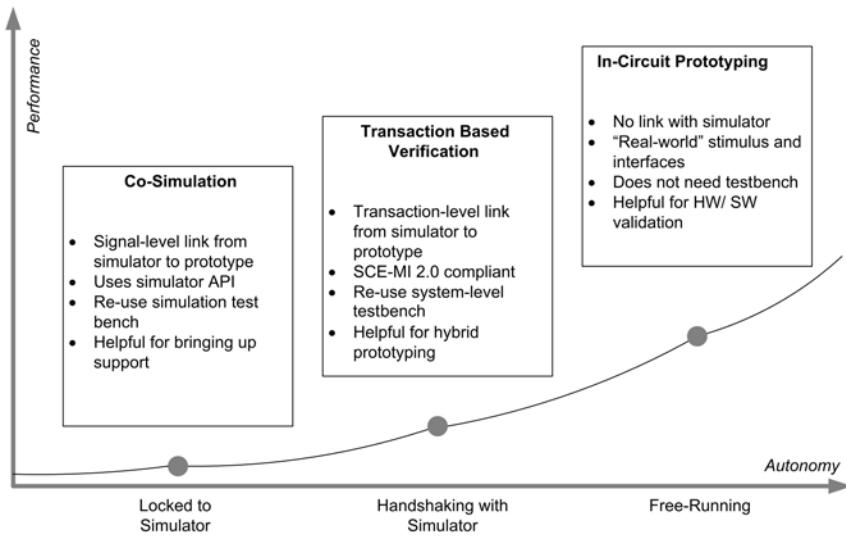
13.1. System prototypes

System prototypes are hybrid verification solutions that incorporate elements of hardware (including FPGA-based systems), software (virtual) models, and traditional EDA simulators. Bringing these technologies together enables designers to mix and match modeling solutions to get the best out of their resources, and to meet their design and verification needs.

There are two core technologies that underpin most system prototyping solutions and enable them to communicate. Co-simulation typically brings together RTL simulation and hardware at the cycle level. The simulator and hardware work in lock-step and communicate through signal-level links. Design teams use transaction-based communication to connect virtual models and hardware using abstracted messaging.

Figure 149 compares co-simulation, transaction-based verification and in-circuit prototyping in terms of relative performance and autonomy. Linking the prototype to a simulator allows the testbench to control the hardware, but at the expense of reduced performance compared with the free-running prototype.

Figure 149 Comparing three approaches to linking FPGA prototypes



13.2. Required effort

Unfortunately, we cannot just plug a prototype into a simulator and expect it to work. Getting a hybrid verification system to work is not always a push-button exercise, although tool environments will automatically generate some of the code that is needed. On top of having code for interfaces, designers have to perform some design conditioning to enable co-simulation and transaction-based verification.

If a design team knows that it will use its prototype in a hybrid verification environment before we start to implement it, the incremental effort required is not that great. It's difficult to quantify the extra design work needed because it depends on many variables. Preparing a design for co-simulation is a question of setting a few constraints within the tool environment. The effort required to enable a design for transaction-based verification depends on whether transactors are already available or need to be created. It is probably reasonable to add 10% to 15% to the "time to prototype" effort in order to ready the FPGA for transaction-based verification.

In order to decide whether the incremental effort is worth it, designers need to know what they can achieve with a hybrid verification environment.

13.3. Hybrid verification scenarios

There are several use scenarios for hybrid verification. We have checked that the scenarios mentioned here are real, not hypothetical. Synopsys engineers have talked to many design teams about their prototyping plans, and this is what they are already doing, or planning to do in upcoming projects.

Design teams often have pre-existing RTL from legacy projects, or have acquired RTL IP for parts of their systems. Reusing existing RTL and targeting it to an FPGA prototype, then combining the FPGA prototype with a virtual model, enables design teams to quickly create a full system model.

Using a pre-existing virtual model for a complex processor core, while using FPGAs to model peripherals, enables design teams to accommodate complex processors and have them run faster than if they were implemented in an equivalent FPGA model.

Sometimes, design teams start out with a virtual model of the whole system. As RTL becomes available or matures, they can replace parts of the virtual model with the new RTL and re-run the tests. If runtimes are short, the design team will re-test the RTL using a normal simulator. Otherwise they will use the hardware.

Using FPGAs with daughter cards or plug-in boards enables the hybrid model to access real-world IO. For certain repetitive tasks it may be beneficial to replace real IO with virtual IO – for example, when it is important to replicate exact inputs for regression tests. This removes any uncertainty to do with using real-world data, since it guarantees an absolute repeat of previous stimulus conditions.

By keeping the FPGA prototype remote and providing software developers with access via their desktops, they can have all the benefits of access to real-world IO, but within a familiar development environment (i.e., keyboard and screen).

For many design teams, the benefits of some or all of these use modes make the investment in enabling their FPGA prototypes for hybrid verification more than worthwhile. We will look again at each of these scenarios in more detail later in this chapter.

13.4. Verification interfaces

To enable the use scenarios outlined above, we need to interface the various parts of a hybrid verification solution and bring together simulators, physical hardware and virtual models. We shall now explore some of those interfaces.

13.4.1. Interfaces for co-simulation

To enable co-simulation, we need to provide a cycle-accurate bi-directional link between a high-performance RTL simulator and an FPGA-based prototyping system

Ideally, we want the link between the technologies to be easy to set up and require no changes to the design itself. Performance of the interface is important, and ideally the link will offer good debug features.

To avoid confusion, it is worth comparing the use of FPGA hardware linked to simulators and a similar technology, known as “hardware in the loop” or HIL. HIL replaces a block in a simulation model with a faster piece of hardware, usually implemented in an FPGA. The aim is to use the FPGA to speed up simulation of the algorithmic model. The FPGA is programmed with some automatically generated code, which is often only functionally equivalent to the simulation model that it replaces and will not be cycle-accurate. The initial design in an HIL approach is often an algorithm modeled in a simulation tool, such as Matlab® or Simulink® and there is no reference to the RTL that will be used in the SoC implementation.

In contrast to HIL, the aim of linking FPGA-based prototyping to simulation is to use the same RTL as the SoC implementation in order to check the validity of the FPGA implementation, possibly with some speed-up of runtime, or to allow system-wide prototyping before the whole RTL is available.

13.4.1.1. Example: HDL Bridge

It is useful to look at an actual co-simulation interface by way of example. HDL Bridge is Synopsys’ proprietary co-simulation link. It provides a bi-directional interface between Synopsys’ RTL simulator (VCS®), and the FPGA-based prototyping system (Synopsys CHIPit® or HAPS® prototyping system). The CHIPit tool can automatically prepare the infrastructure to communicate between the simulator and hardware based on the RTL provided by the user. CHIPit creates wrappers without disturbing the original design – a synthesizable wrapper for the hardware (either Verilog or VHDL) and a separate wrapper for the software simulator.

This environment is easy to set up and allows seamless integration with VCS simulation. Once constrained, generating the wrappers is very fast as we need only provide the top-level design description, define the clock and reset polarity.

The interface enables comprehensive debug by allowing the simulator to monitor all internal registers of the design under test (DUT) in the FPGA.

Figure 150 shows how the HDL Bridge is partitioned so that the non-synthesizable part of the interface remains in the simulator, while the synthesizable code is

implemented in hardware. The wrappers get data from the hardware and software and give it to the simulator's PLI interface as it makes PLI calls. The simulator testbench controls the design.

Figure 150: HDL Bridge co-simulation partitioning

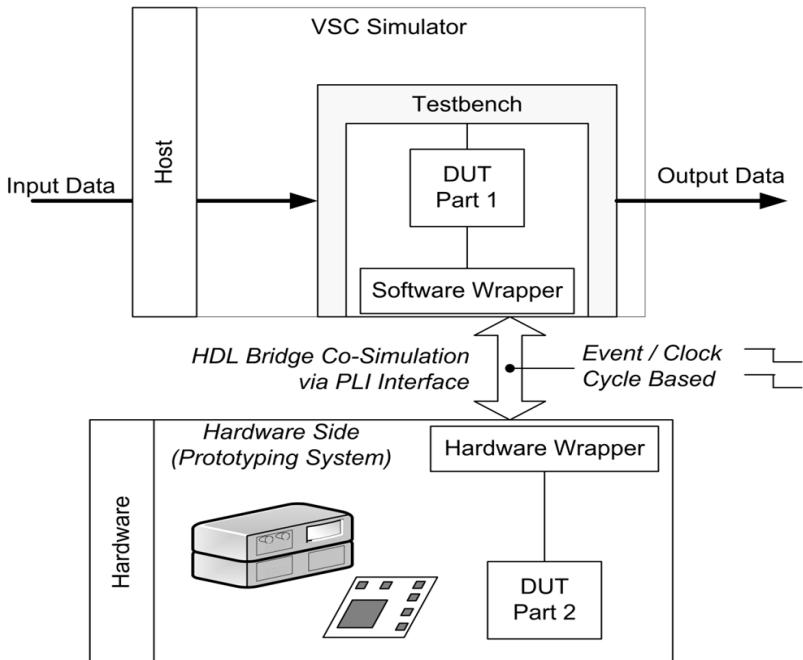
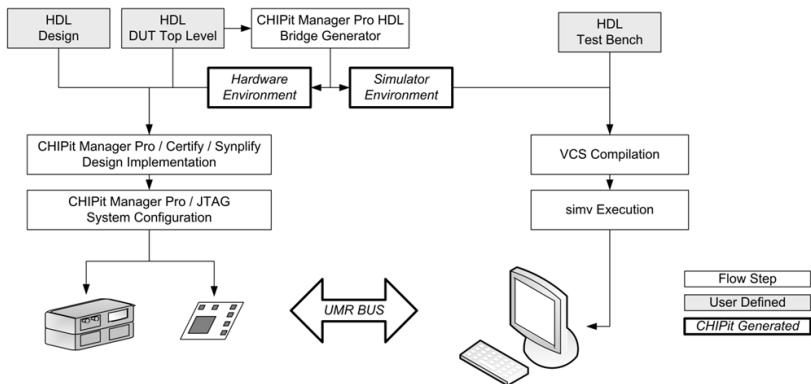


Figure 151 shows the overall design flow for the Synopsys environment and highlights the user-defined inputs and the automatically generated code.

Co-simulation suffers from a fundamental limitation: because the simulator is only capable of running at a speed of a few kilohertz, and the hardware runs in lock-step, it is not possible to use real-time interfaces or components that have a minimum frequency such as technology-dependent primitives including PLLs, DCMs and IP PHYs. We have covered the issue of addressing FPGA minimum frequency in chapters 7 and 9, which readers should consult for recommended design strategies.

Figure 151 HDL Bridge co-simulation design flow



Recommendation: design teams can save themselves time by thinking ahead and anticipating issues. Planning the clock structure to support a low-frequency clock will make the task of setting up for co-simulation easier

Another issue that designers must watch out for is the trade-off between debug and performance. HDL Bridge allows designers to monitor the internal registers in the FPGA, as they increase the number of signals to monitor, the performance of the interface decreases.

In summary, co-simulation can take advantage of any simulator's API in order to push stimulus to the hardware and receive its responses. It is easy to set up and no design changes are necessary. The tool takes care of preparing the infrastructure to communicate between the simulator and hardware once we have created a top-level description and defined the clock and reset signals.

13.4.2. Interfaces for transaction-based verification

In co-simulation, the simulator is controlling the design in hardware, while in transaction-based verification, the DUT (in hardware) and simulator (or software application running on a host machine) communicate by passing messages or transactions. This requires an abstracted bi-directional link between virtual models and the FPGA-based prototyping system.

Communicating through transactions and the use of transaction-level models enable faster simulation and easier debugging than co-simulation. Using transactions, design teams can focus on the function and behavior of their systems and get that right before they concern themselves with implementation. They can also define the verification scenarios that they want to cover more quickly and easily, because they are using software running on the processor.

13.4.3. TLMs and transactors

The SystemC transaction-level modeling standard (TLM-2.0) defines two coding styles: loosely timed (LT) and approximately timed (AT). TLM-2.0 models, which themselves include transactors, enable efficient message-based communication for exploring the system at a high level of abstraction when written in an LT coding style. A model that includes an LT transactor is relatively fast to develop because the transactor simply deals with reads and writes to memory locations.

However, in the hardware world we cannot ignore timing forever. That is why TLM-2.0 allows designers to create transactors with timing annotations by writing models using AT coding styles. Using these, design teams can perform tasks such as estimating software performance and analyzing different architectures. Developers can create AT models relatively quickly.

As designers refine their systems down to real hardware, they need to add even more timing detail. To enable this we need transactors that can convert from the function level to and from the signal level. For example, taking TLM-2.0 transactions to, for instance, AMBA® AHB/APB/AXI™ interconnect signals, and also handling side-band signals such as reset inputs and interrupt request outputs. These transactors have a cycle-accurate interface to RTL on one side, and a transaction-level interface on the other side.

Design teams can achieve high-speed transaction-based verification by processing the compute-intensive part of the transactor in hardware, rather than using software on the host workstation. This is possible if the transactor is coded as a synthesizable state machine, or bus functional model (BFM), which receives messages and converts them into signals that are connected to the design ports or internal buses. Developing transactors for complex interfaces such as AXI can be time-consuming, so reuse is extremely desirable.

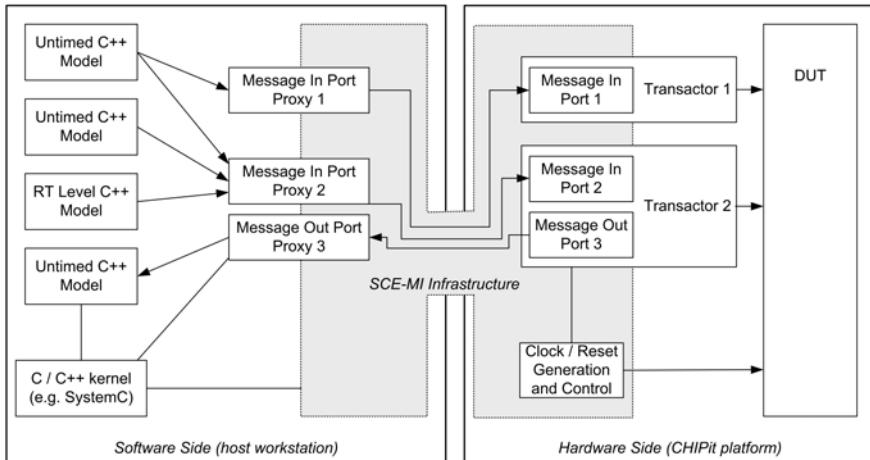
The transaction-level interface between the synthesized and simulated parts of the design is made up of fixed-width unidirectional input and output ports. An input port sends messages carrying transaction-level information from the simulated testbench layers to the hardware-assisted layer. An output port sends messages from the hardware-assisted layer to the simulated layers. There can be any number of input and output ports. However, the hardware platform may place some restrictions on their number, width or total width.

13.4.4. SCE-MI

The Accellera standards organization approved version 2.0 of the Standard Co-Emulation Modeling Interface (SCE-MI) in 2007. SCE-MI provides a multi-channel message-passing environment for transaction-based verification. The standard defines how messages can be sent between software and hardware.

SCE-MI enables design teams to link transaction-level models to hardware accelerators, emulators and rapid prototyping platforms by interconnecting untimed software models to structural hardware transactor and DUT models (Figure 152). It provides a transport infrastructure between the emulator and host workstation sides of each channel, which interconnects transactor models in an emulator to C/C++/SystemC (untimed or RTL) models on a workstation.

Figure 152 High-level view of runtime components in SCE-MI based co-modeling



In SCE-MI version 1.1, the transport infrastructure provides interconnections in the form of message channels that run between the software side and the hardware side of the SCE-MI infrastructure. Each message channel has two ends. The end on the software side is called a message port proxy, which is a C++ object that gives API access to the channel. The end on the hardware side is a message port macro, which is instantiated inside a transactor and connected to other components in the transactor.

A message channel is unidirectional – either an input or an output channel with respect to the hardware side. However, a message channel is not a unidirectional or bidirectional bus in the sense of hardware signals, but resembles a network socket that uses a message-passing protocol. The transactors are responsible for translating the message-passing protocol into a cycle-based protocol at the DUT interface. They decompose messages arriving on input channels from the software side into sequences of cycle-accurate events which can be clocked into the DUT. In the opposite direction of informationflow, transactors recompose sequences of events

coming from the DUT back into messages to be sent via output channels to the software side.

Furthermore, the SCE-MI 1.1 infrastructure provides clock (and reset) generation and shared-clock control using handshake signals with the transactors. This way the transactors can freeze controlled time (by suspending the clocks) while performing message composition and decomposition operations.

SCE-MI 2.0 adopts SystemVerilog's direct programming interface (DPI) function-call model and adds additional features to enable transfer of bi-directional variable-length data between the DUT and the software testbench using just four functions and pipes.

Table 31: SCE-MI 2.0 hardware and software calls

	Hardware call (h/w to s/w)	Software call (s/w to h/w)
Data transfer	Import function	Export function
Streamed data transfer	Send pipe	Receive pipe

In case of SCE-MI 2.0, the clock delivered to the SCE-MI transactors is controlled by the infrastructure and therefore user intervention is not required to stop and start these clock signals.

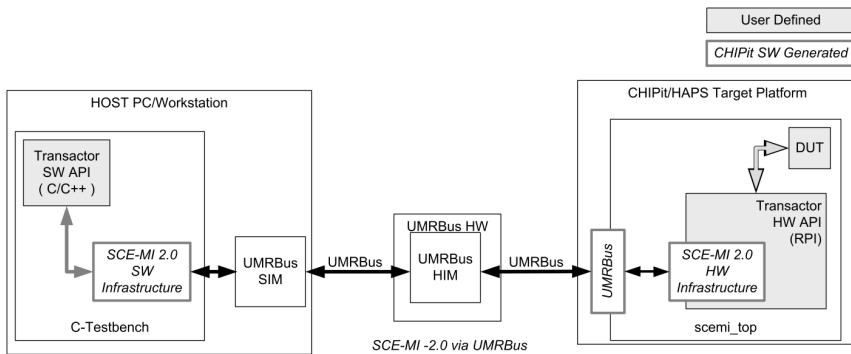
Recommendation: if there is a huge amount of data to be sent, then it is necessary to use pipes. We can use data transfer functions if there are only control signals or small amounts of data to be sent. If data transfer functions are used for large amounts of data the handshake overhead can exceed the size of the packet sent.

13.4.5. SCE-MI 2.0 implementation example

In this example, illustrated in Figure 153, the transactor and infrastructure communicates with hardware through the UMRBus®. Synopsys' CHIPit tool uses the user-defined transactor description to create the hardware infrastructure for the DUT, and also writes out C/C++ files for the software environments. The communication between software and hardware uses its own clock domain (SCE-MI clock), which is independent of the DUT's clock domain. During data transfer between the host software application and hardware via SCE-MI, the SCE-MI controlled clocks are stopped on the hardware. SCE-MI infrastructure releases the controlled clocks once the data transfer is complete. Users do not need to take any

special steps to manage this – the control clock handling procedure is part of the SCE-MI 2.0 infrastructure.

Figure 153: SCE-MI 2.0 co-emulation overview for CHIPit®/HAPS®



Another way to use SCE-MI is to communicate with a simulator incorporating SystemVerilog testbenches. The SCE-MI 2.0 standard defines a way to communicate between software and hardware. If users need to communicate with a simulator within a SCE-MI 2.0 environment, Figure 154 shows how the simulator's SystemVerilog testbench can talk via SystemVerilog's DPI calls to a C-environment, and the very same C software can talk to the connected hardware via DPI-like SCE-MI 2.0 function calls.

Figure 154: SCE-MI communication through DPI

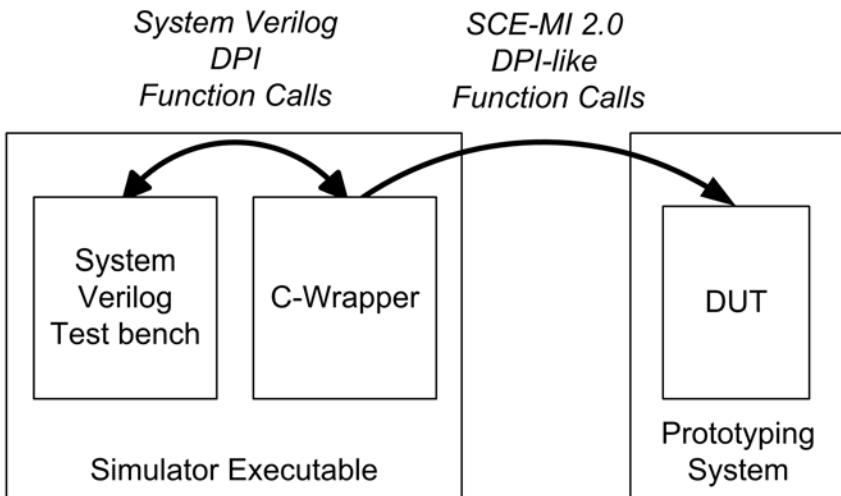


Figure 154 shows, on the right, a hardware system with a SCE-MI DPI function. The simulator executable (left) includes a C-wrapper file which has SCE-MI DPI functions, as well as a pure SystemVerilog communication channel that enables communication between the C software and the SystemVerilog testbench. The VMM hardware abstraction layer (VMM HAL), which is described below, is an example of this approach.

13.4.6. VMM HAL

The Verification Methodology Manual (VMM) defines a methodology for verification that has become widely adopted by design teams. The VMM HAL is a VMM application that includes a class library to support transaction-level co-emulation between a hardware-accelerated design and a VMM-compliant testbench running on a SystemVerilog simulator, such as Synopsys' VCS functional verification simulator.

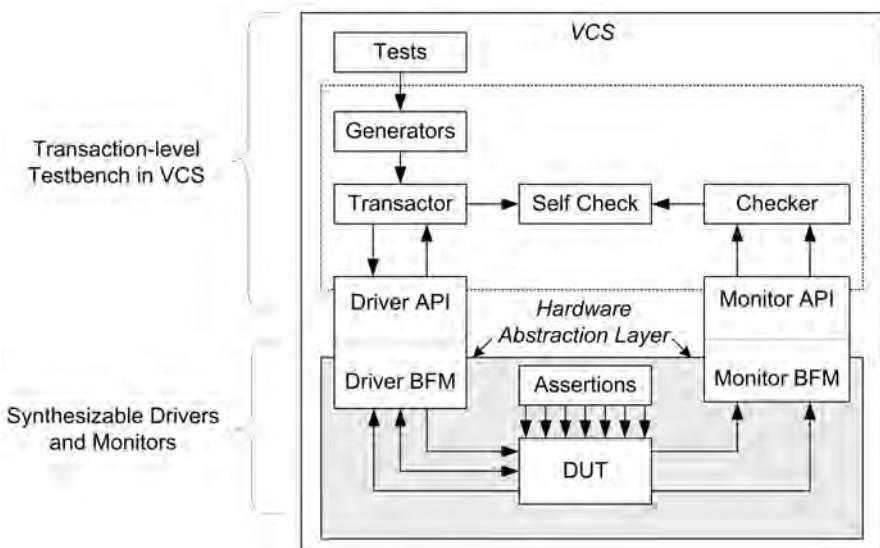
The hardware abstraction layer enables designers to use different hardware acceleration platforms with the same constrained-random testbench that is used in simulation-only environments. Testcases and DUT can target different hardware platforms without any modifications. Figure 155 gives a very brief overview of the VMM approach and we see that there is a verification loop driven by high-level, often object-oriented code which hardly resembles a testbench in the traditional

sense. The loop would normally continue via transaction-level drivers into the DUT and out through transaction-level monitors to complete the loop with some comparisons and a scoreboard of results passed.

With the HAL, the transaction-level drivers and monitors are replaced with SCE-MI2.0 transactors and the DUT is the FPGA-based prototype containing a version of the SoC design.

The VMM HAL application also contains a purely simulated implementation of the hardware abstraction layer that allows the testbench and testcases to be developed and debugged with the DUT entirely within the same simulation, without modifications, and without access to the hardware emulator.

Figure 155 HAL-compliant transaction-level testbench



The VMM HAL environment can target hardware or simulation at runtime by using a simple simulator command-line switch. The environment includes drivers and monitors, which are software-software transactors in a normal VMM environment. VMM HAL transactors for hardware must be synthesizable.

Some companies are starting to build up VMM HAL transactor libraries. A key benefit of the VMM HAL is that it maximizes reuse of testbench code. A designer does not need to modify the top layers of the verification environment (such as tests, generators and transactors), only the lower-layer monitors and drivers need to be made HAL-compliant.

13.4.7. Physical interfaces for co-verification

Physical interfaces for co-verification are those implemented in the prototyping environment to support communication between a host system and hardware. The physical layer can use any transport mechanism, and design teams can opt to use a standard PC interface like PCIe. Whatever physical interface they settle on, ideally it should offer ease of use for faster validation of the hardware prototype, better design debugging and easy prototype configuration. Support for advanced use modes will also be beneficial, especially support of: standard APIs; RTL-based co-simulation and debugging; accelerated transaction-based verification; and connecting to and co-simulating with virtual prototypes. Interfaces that enable rapid initialization of the system and remote access and management of the FPGA-based prototype allow design teams to get the most use from their prototypes.

Synopsys has designed the UMRBus physical interface specifically to provide high-performance, low-latency communications between a host and Synopsys' FPGA-based prototype platforms. We previously introduced UMRBus in chapter 11 in the context of debugging. To recap, UMRBus is a high-speed connection between the host workstation and the prototype that provides parallel access to all FPGAs, board-level control infrastructure and memories (internal and external) on the Synopsys FPGA-based prototyping platforms.

13.5. Comparing verification interface technologies

In order to compare different verification interfaces, Synopsys has created a demonstration benchmark based on an image processing sub-system. The demonstration system uses a JPEG algorithm to compress an image, which is stored on a host computer. We first simulated the RTL in VCS running on the host computer before moving the DUT from simulation to a synthesized target running on an FPGA prototype, which enabled us to compare co-simulation with HDL Bridge, SCE-MI and UMRBus.

The comparisons and performance data are summarized in Table 34 later but in the meantime, Table 32 summarizes the comparison between the four different modeling techniques.

Table 32: Summary of performance and benefits of four interface implementations

	RTL simulation	Co-simulation	SCE-MI	UMRBus
Need to change the HDL?	No	No	Yes	Yes
Need to generate wrappers?	No	Yes: HDL Bridge	Yes, tool generated	No
Clock mode	Host controlled	Host controlled	SCE-MI controlled	Free running
Signal visibility	All	Registers and DUT ports	All	All
Trace resolution		Cycle-accurate/event	Variable	Variable
Trigger support		Yes	Yes	Yes
Runtime for JPEG image processing	700s	50s	160ms	70ms

For this particular task, we can see that UMRBus is the fastest implementation. SCE-MI, implemented on top of UMRBus is second fastest thanks to its use of transaction-based communication. Co-simulation using cycle-based interfaces for this task is about 300x slower than SCE-MI, and RTL simulation is over 4000x slower than SCE-MI. While this data helps us to compare the different approaches, it is not possible to generalize. Performance depends on many different parameters, for example, how much computation is happening on hardware and how much is in the simulator, also, what is the amount of data that is being exchanged between different sides of the transactors, and what transport mechanism is used? Of course we will also get different results for different kinds of design.

The UMRBus channel is for the sole use of the co-simulation and SCE-MI implementations. Using an alternative bus standard, such as PCIe, may deliver worse performance as the channel is not a dedicated interface and it may have to handle other traffic as well as the co-simulation data.

As well as considering performance, it is worth comparing the debug capabilities of each approach. UMRBus enables users to interact directly with the design from the software world. Intensive debug activity in HDL Bridge or SCE-MI may reduce overall performance. The number of signals that we can capture in HDL Bridge may be limited.

It is also possible to perform debug by physically connecting to the FPGA using a logic analyzer and physical probes. This allows probing of internal FPGA signals, and it is useful for tracing free-running designs. There are a couple of potential drawbacks with this approach to debugging – there may be a delay on debug traces, and probe signal names may not exactly match the original RTL names.

13.6. Use models – more detail

Having gained a better understanding of the types of interfaces available for co-simulation and transaction-based verification, we look in more detail at the use models we summarized earlier in this chapter.

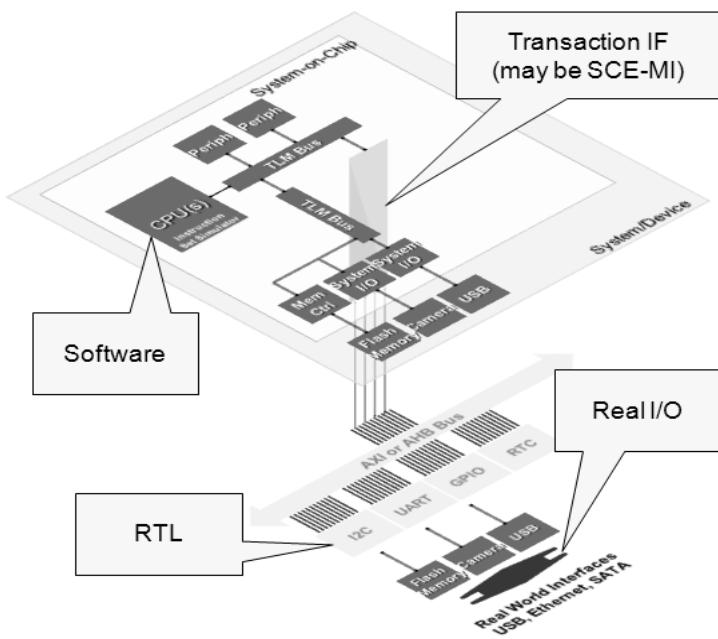
13.6.1. Virtual platform re-using existing RTL

In this case (Figure 156) the project team wants to use a virtual platform, although they are concerned about the overall modeling effort required. Because they need good performance, they decide to instantiate the RTL in the FPGA rather than a simulator. It may be that they are developing a new architecture, so they want to start to develop software before all the RTL is available.

In this case we can provide a virtual platform with, for instance, the latest ARM® core, which will probably be available before ARM has actually introduced it as RTL or physical IP. The core could be modeled in the virtual platform but we may need to integrate it with a high-performance sub-system for which we do not have a system-level model. For example, that might be a high-definition video codec for an imaging sub-system for which we have RTL code from a previous design. We can decide to implement the codec in the FPGA, gaining accuracy and performance, but keep the rest of the design in the virtual prototype, at least until that RTL also becomes available.

It might take two to three months to bring up a new prototype from fresh RTL but to create a virtual platform model from scratch might take even longer but it would still be quicker than waiting for the new RTL. By mixing virtual models and legacy RTL we can have a better chance of bringing up a hybrid system in the quickest time. The other benefit of a hybrid system is that at least part of the design will have cycle accuracy.

Figure 156: Virtual platform re-using existing RTL

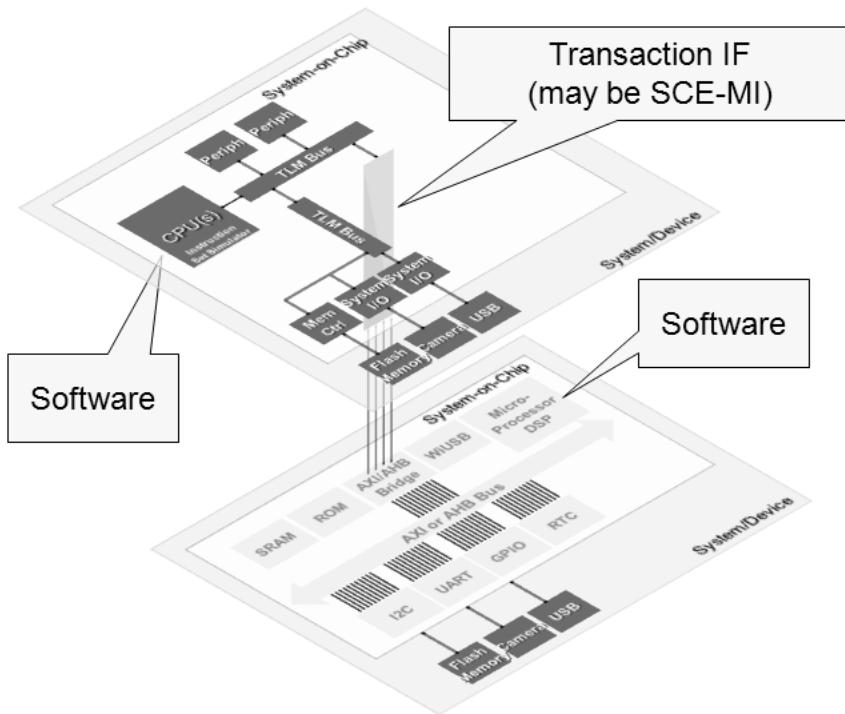


13.7. Virtual platform for software

FPGAs are not optimized for processor implementation, so a design team that wants to use a complex processor, such as an ARM Cortex®-A8 or similar, will find it difficult to instantiate the core in the FPGA and achieve the performance they are looking for.

One option (Figure 157) is to partition the design so that they have a fast ISS running within the virtual platform. They have to partition the design sensibly so that they can perform most of the transactions in software, and only when accessing peripherals should the system access the FPGA. They should ensure that all the local memory (e.g., L3 memory) is also within the virtual platform; otherwise they would see no performance benefit as a result of going across the SCE-MI interface to fetch instructions and data.

Figure 157: Embedded software executed on host



In this example the design team performs native execution of the embedded software on the host PC. Executing software on a workstation with a virtual model processor is often faster than in FPGA prototype. The design team can use the FPGA to maintain the accuracy of accelerators and peripherals.

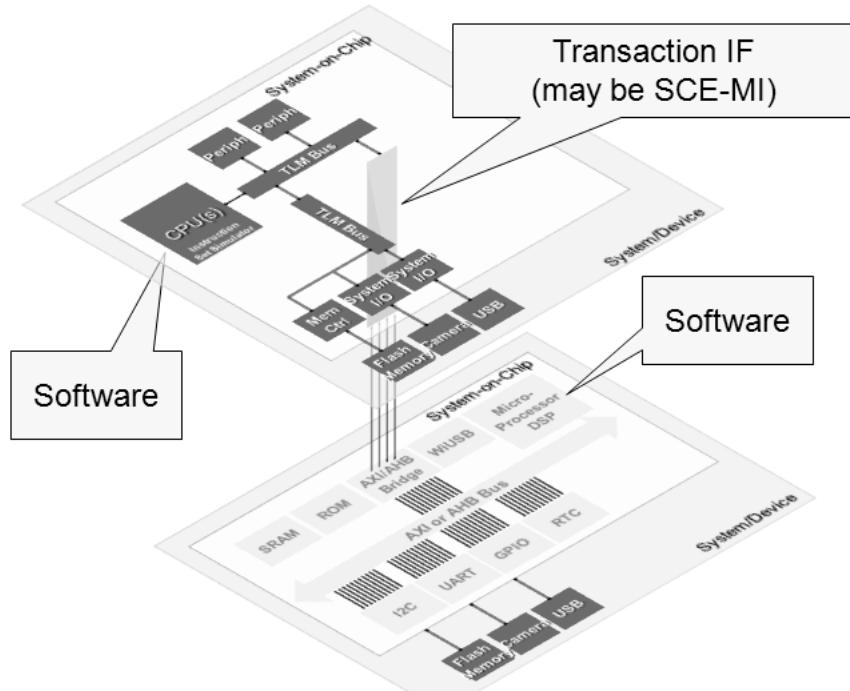
13.7.1. Virtual platform as a testbench

Design teams tend to use this approach (Figure 158) when they have kicked off the project with some pre-RTL software development. They may have actually started to write test cases to verify some of the blocks in the virtual platform. This is an approach that we have used within Synopsys to verify USB software.

When the RTL becomes available, they can replace the TLMs with the RTL instantiated in the FPGA and re-run the same software verification tests against the RTL to check that they pass, or otherwise refine the test cases. While some of the IP blocks that they have modeled may have only partial functionality, they will have full accuracy (including the introduction of cycle-accurate timing) in the RTL. By

abstracting some of the accuracy at the outset, the developers can get started on the software development sooner.

Figure 158: Virtual platform acts as a testbench for RTL in FPGA



Using the virtual platform as a testbench for the FPGA prototype avoids duplication of effort by making better use of the work that went into system-level development. It also enables a design team to compare results in simulation against results in hardware, to verify the flow into hardware by repeating a set of known stimuli, to provide feedback to the verification plan for quality assurance purposes and to run regression tests very efficiently.

13.7.2. Virtual and physical IO (system IO)

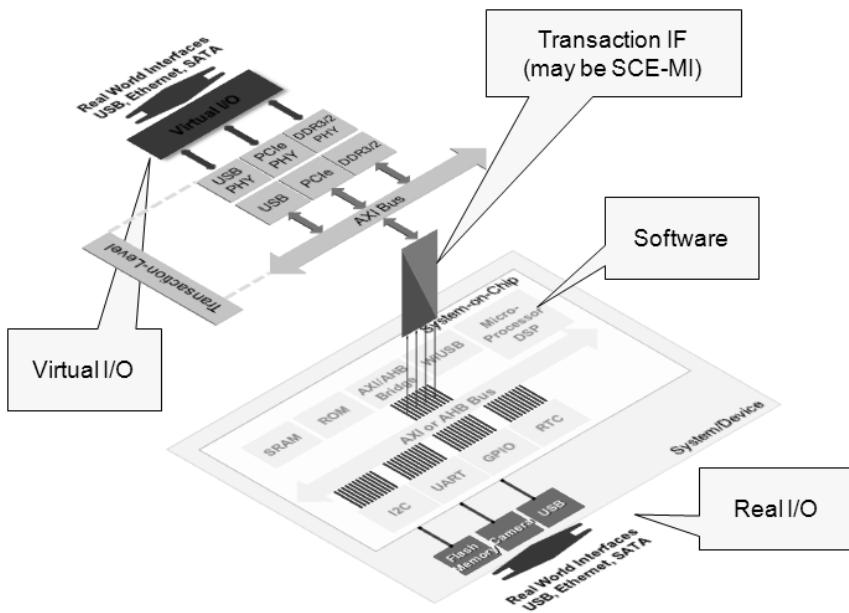
In this scenario (Figure 159) the design team has a certain amount of virtual real-world IO that it can support from the virtual platform, but this situation is not as good as having a real physical interface. To reduce risk, we need to verify the design in the context of the system it lives in, by connecting it to the real world. Any new interface standard would benefit from this approach.

Design teams want to implement standards before silicon is available. We can support this requirement by using a plug-in board (physical hardware) that would confer the ability to use a standard virtual platform to bring up an OS, like Linux. In this case we would have a new IP block and would be able to interface it to the real world for test purposes.

Take the example of integrating a camera in a phone applications processor chip. Taking this approach we could link the baseband design to a camera on a workstation and make the software believe that information is coming from the real camera in the phone. This allows the use of real-world stimuli when appropriate.

For some applications it may be useful to create a GUI or some other interface to allow the design team to interact more naturally with the virtual environment. For example, “Can the phone receive a call while playing a game and downloading email?” is the kind of scenario that using a GUI will help to verify.

Figure 159: Virtual platform with a physical connection to the real world



13.7.3. Virtual ICE

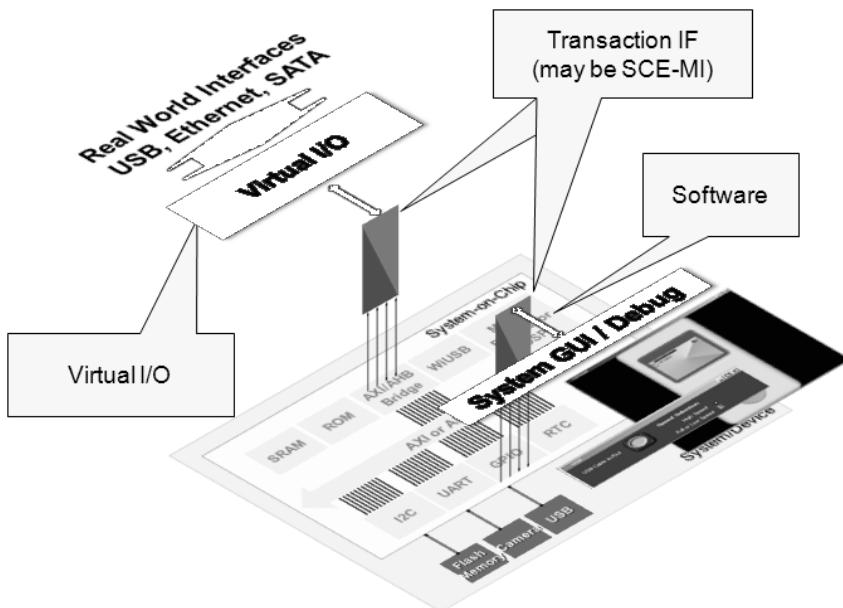
Sometimes software developers do not want or need to see a board on their desks, yet they do need access to the functionality that the DUT provides in order to write code. This approach (Figure 160) provides a virtual hardware capability and helps

to keep software engineers out of the lab and in the comfort of their familiar development environment – at a PC.

With a virtual ICE the design team can give remote access to the software developers – virtual ICE is the remote desktop solution. They can have a very small element of a virtual platform running on each of the software developer's desktops.

The idea is that the design team has an FPGA model that includes interfaces like LCD and UART. They direct those interfaces back up to a virtual platform on a PC so that they can have an LCD instantiated on the user's desktop. They would redirect the hardware traffic back across SCE-MI, giving a software virtual interface to the hardware. Rather than connecting the FPGA to a physical LCD or physical UART, they would model those interfaces within the virtual platform. They implement the majority of the system in FPGAs, and provide virtual interfaces to some of the physical hardware ports.

Figure 160: Platform provides links to designers at their desks



This approach is especially beneficial if the prototype has to remain in the lab because of its size or fragility. The disadvantage is that simulation performance will be worse over the network than having a high-speed connection direct to the prototype.

13.8. System partitioning

Getting the hardware-software partition in the right place is critical to achieving good system performance. There are a number of issues that designers must consider, as well as some practical guidance to follow.

First, there are certain fixed constraints. Some parts of the system – for example, the testbench – may just not be synthesizable, so the design team cannot place them in hardware. They may also be constrained by design size and need to map the design to multiple FPGAs. Remember that high utilization of the available gates leads to longer implementation times, especially because of place & route.

For parts where they have choice, design teams need to decide where to put the bridge between hardware and software. They can only make a cut where they can use a transactor that already exists, or that they can easily obtain. This tends to favor inserting the partition at well-defined industry standards, such as on an AHB™ bus interface. Making a cut at some arbitrary point means that the design team has to come up with a way of modeling it, which can create problems.

Once they have considered the constraints, the design team must analyze the design to understand where they can make the cut in order to reduce the communication between the simulator and the hardware. To maximize the chances of having the FPGA accelerate the design, ideally they need to have computation in the hardware dominate communication between the simulator and hardware.

Whether a design team will see their design accelerated depends predominantly on the traffic across the interface between hardware and software. If the traffic is characterized by a few control signals, the design team will likely see a huge speed-up. On the other hand, heavy interaction between the simulator and DUT may yield a small speed-up, or none at all. Table 33 shows how applying Amdahl's law helps to predict simulation acceleration.

Recommendation: if the aim is simulation acceleration, consider where computation is happening. Move more and more components into the DUT, if possible synthesize the testbench so that everything runs in the DUT.

Table 33: Ahmdahl's law predicts simulation acceleration

Share of simulation effort		
Testbench	90%	10%
DUT	10%	90%
<i>Maximum acceleration factor</i>	10%	900%

Sometimes, design teams choose to successively refine their partitions by moving more and more into the DUT, as the RTL becomes mature. Not committing untested code to the FPGA helps them to manage risk. For maximum performance they can move across all synthesizable parts of the testbench.

Recommendation: if the DUT uses multiple FPGAs, dedicating the simulator interface to just one of the FPGAs will help improve performance.

13.9. Case study: USB OTG

This case study shows how Synopsys combined a virtual platform and FPGA prototype to create a system prototype for a USB on-the-go (OTG) core.

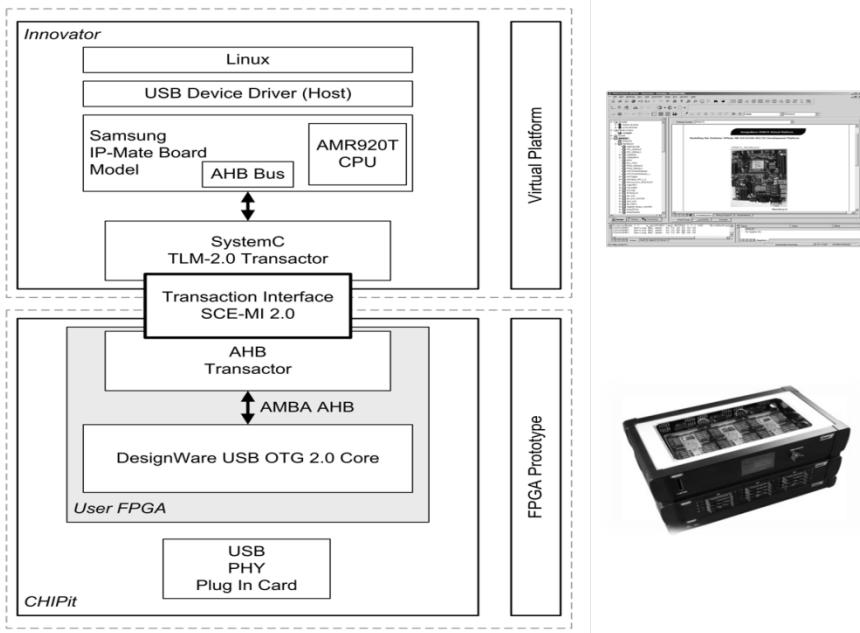
13.9.1. USB OTG System overview

The system (

Figure 161) consists of a virtual platform modeling a Samsung system-on-chip supporting LCD, touchscreen, DMA, and physical Ethernet, running an unmodified hardware Linux image. The virtual platform connects via an AHB bus with transactors over the SCE-MI 2.0 interface to the USB 2.0 OTG core running in the FPGA prototype.

We can connect a USB memory stick containing pictures to the system prototype by using a daughter card. The virtual platform controls the memory stick which provides access to the images. Users can debug at the hardware-software interface with the software debugger and the hardware debug environment.

Figure 161: CHIPit Innovator system prototype



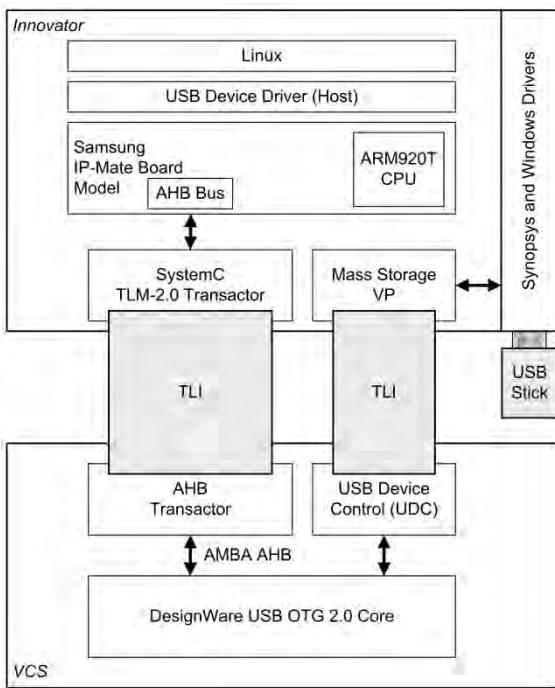
13.9.2. Integration use models

The system prototype enables various use models. One of the key benefits is achieving a significant speed-up over pure RTL simulation – in this case by a factor of 6x. The debug insight and controllability at the hardware-software interface also boosts productivity. The environment supports features such as hardware breakpoints, including the ability to pause and single-step the entire simulation. System visibility and logging is only constrained by the host PC memory and storage capacities.

13.9.3. Innovator and VCS

We can integrate the Innovator model with VCS through SystemC by using PLI TLMs. We can partition the system so that the USB OTG RTL description runs within VCS. The disadvantage of this configuration is that VCS cannot physically control the memory stick. However, we can work around this by reflecting the accesses back up to the software (the virtual prototype), and then connecting to the USB stick.

Figure 162: Innovator and VCS



In practice, because the RTL is running much slower than the FPGA it is very difficult to control a physical memory stick. That is why most design teams would choose to use an FPGA prototype rather than co-simulation. There are, however, advantages of using a VCS solution, including its superior analysis capabilities and having better insight into the behavior of the core running on VCS – more so than a designer would have with CHIPit, for instance.

13.9.4. Innovator and CHIPit or HAPS

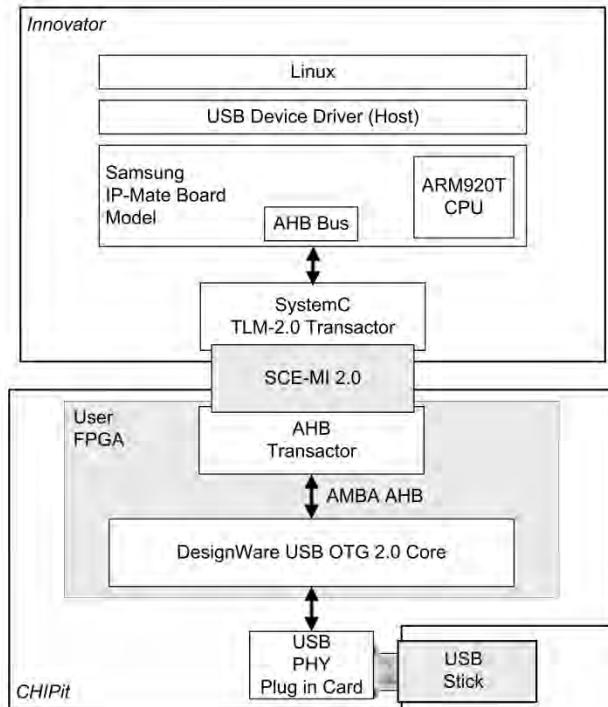
This use case (

Figure 163) consists of Innovator connected to CHIPit or HAPS via SCE-MI.

On the software side is a library that allows us to send transactions (which may just be reads or writes) across SCE-MI. In the physical prototype we have a

synthesizable transactor that we have implemented in an FPGA, which can then interact with the RTL for the USB OTG.

Figure 163: Innovator and CHIPit (or HAPS)



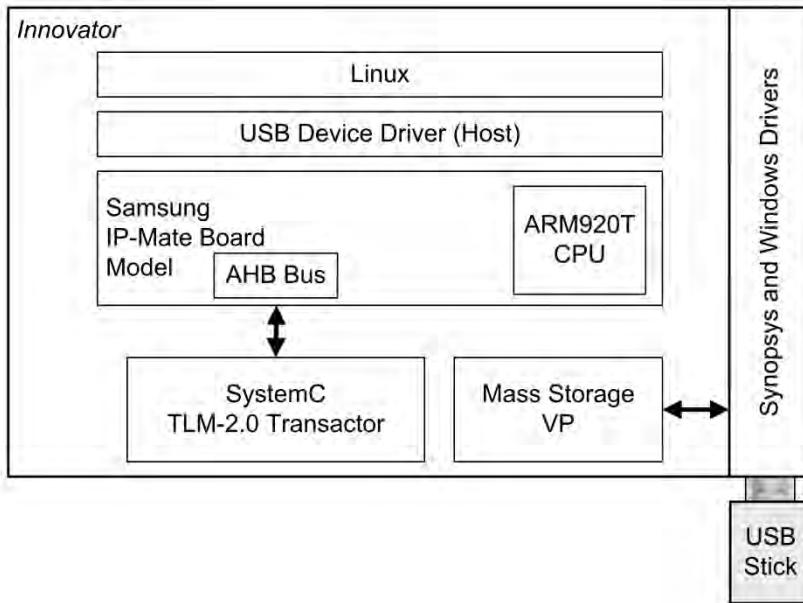
The advantage of having the transactor instantiated on CHIPit is that we can have a plug-in card for the physical USB interface, so we can control a physical memory stick or some other physical real-world device. Because the function is in hardware, it is fast enough to do that. In some cases, this may also differentiate this use case from the pure virtual prototype use case.

13.9.5. Virtual platform

The advantage of the pure software virtual prototype is that at the pre-RTL development stage, there may not be any RTL available, so there is no other easy way to do co-simulation or transaction-based verification. That is basically why

software virtual prototyping complements the hybrid or hardware approaches: developers can start to develop their software, pre-RTL, on a software virtual prototype assuming that they have access to models. It will take a certain amount of time to develop new IP blocks. Despite that, typically we can have customers engaged in software development some 9-18 months before tape-out.

Figure 164: Pure software virtual platform



Performance Comparison Table 34 shows performance figures for booting the system and for performing the mount, copy and unmount operations.

Table 34: Summary performance and characteristics for USB OTG example

	RTL	RTL + Virtual Platform	Virtual Platform	System Prototype	Pure Hardware
Accuracy	Cycle accurate	Adjustable, depending on partition	Transaction functionally accurate	Adjustable, depending on partition	Cycle accurate
Availability	During RTL development	Prior to RTL, supports legacy	Prior to RTL depending on models	Prior to new RTL, supports legacy	At end of RTL development
Linux boot	-	86s	29s	43s	12s
Modprobe	-	702s	23s (31x)	61s (12x)	10s (70x)
Mount	-	-	15s	22s	< 1s
Copy	-	-	4s	3s	<1s
Unmount	-	-	<1s	9s	<1s
Scaled	0.03125x	1x	225x	~30x	625x

The pure virtual platform performance was actually faster than the system prototype because in this particular design there was a lot of traffic going across from the virtual prototype to the RTL, in order to process interrupts. USB is not the best example to demonstrate performance acceleration via hardware because of the number of interrupts that the core generates and the processor needs to service. In fact, the USB generates a start-of-frame interrupt in high-speed mode every eighth of a millisecond.

We had to look at ways of optimizing this design to stop the interrupts swamping the bandwidth, with the consequence of a decline in performance. We did quite a lot of work to boost the performance of the virtual platform. Partitioned properly, something like a video codec would see better performance with the system prototype than with the virtual platform.

The authors gratefully acknowledge significant contribution to this chapter from

Rajkumar Methuku of Synopsys, Erfurt, Germany

Kevin Smart of Synopsys, Livingstone, Scotland

This chapter will try to predict how chip and system design will look like five years from now and how prototyping will be impacted, and vice versa. We will again consider prototyping in its many forms and FPGA-based prototyping within that framework.

14.1. If prediction were easy... .

. . . then flip-flops wouldn't need set-up time (*pardon our little engineering joke*).

As they say, predictions are difficult, especially about the future. However, in the case of prototyping, the general direction for the next few years can be predicted pretty well. We can look back at chapter 1 and recall the industry trends for semiconductor design and our 12 prototyping selection criteria. We can then use these insights to predict how our need for prototyping might evolve in the coming years. Of course, different SoC designs in various application areas have different needs, so we shall start by examining the specific needs of three important application areas.

14.2. Application specificity

When looking into the crystal ball it becomes clear that chip development needs are highly application-specific. The application areas targeted by a design will, to some extent, govern its development methods and timescales. However, the common denominator will always be the software, which increasingly determines system functionality and changes the very way that hardware is designed in order to efficiently run that software.

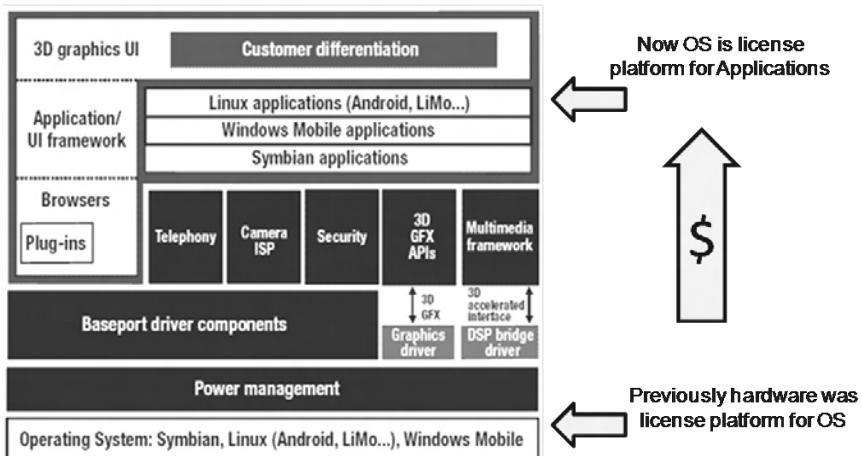
Today – in late 2010 - we have already reached a point at which the application domains significantly determine chip design requirements, most importantly on the path from idea to implementable RTL combined with software, and subsequently its verification. Given that different application domains use different IP and interconnect fabrics as well as have different software development requirements, design flows are increasingly become even more tailored to particular application domains.

Let's compare the trends in three typical domains, specifically wireless/consumer, networking and automotive.

14.3. Prototyping future: mobile wireless and consumer

The user experience in wireless and consumer applications is today already mainly determined by applications running in higher-level, hardware-independent development platforms like Java™ and specific SDKs. Figure 165 shows the trend from proprietary operating systems towards open operating systems, which encourage application developments mostly from third-party developers. Since the introduction of online application stores, some consumer hardware has become the

Figure 165: The shift from proprietary operating systems to application driven design



delivery vehicle for distribution of applications, which are controlled by the operating system that enables them. Application developers are offered a distribution vehicle and in exchange they pay a percentage back to the environment's proprietor. The result is that, in a unique way, the OS providers have found a vehicle to monetize the end applications rather than hardware platforms.

In the past, the operating systems themselves used to be quite profitable licensing businesses then in 2008 there were some sudden changes with handset operating systems being acquired by major vendor, while conversely we saw the release of the Android operating system. Several handset providers also released their own proprietary operating system platforms. At the time of writing in 2010, Microsoft®

Windows Mobile™ has become the only widely-used operating system that still charges license fees to mobile handset manufacturers.

Instead of making the OS a “licensable product,” the new model is that the handsets and the embedded OSs running on them have now become a channel to provide applications of astonishing depth and variety. We estimate that users have a choice of more than 500,000 different applications across the various operating systems and platforms, accessible through various online application stores. For mobile applications this defines a fundamental shift in where the business value lies and it is unlikely to be reversed.

What does all this mean for chip design and prototyping? The effect on development is that because the value is moving into applications, software will increase even further in importance and its development needs will take precedence. In effect, the software will increasingly govern how the hardware is designed. In the case of mobile wireless and consumer applications this means that hardware developers need to provide fairly generic execution engines as early as possible and independent from the actual hardware. For our 12 prototyping selection criteria, this means that replication cost and time of availability will be of highest importance. The sheer number of application developers will require a very inexpensive way to develop applications, which will push more capabilities into SDKs. Time of availability will be important and in a sense hardware and software development will increasingly be done upside down – with the software being available before the hardware and the hardware designed to execute OS-based software in the most efficient way.

While the trend to isolate software development from hardware effects using hardware abstraction layers and OSs will strengthen even further, user expectations for high quality applications will grow and as such application verification will also gain importance.

Future SDKs will have to provide some of the application verification capabilities which already exist today for other software development environments like virtual platforms and host development environments. For example, software memory checking is a well-known technique in the host workstation space and quality verification tools such as Valgrind, Purify, BoundsChecker, Insure++, or GlowCode are part of any industry-strength software design flow. However, these types of tools are not generally available or widely used in the embedded world. SDKs and virtual platforms are the appropriate prototyping areas to which these capabilities should be added.

There will still be a need for FPGA-based prototyping for the lowest levels of the software stack where speed and accuracy are needed at the same time. In addition, the needs of the hardware platform do not become any more relaxed. For example, the leading platforms will be low power and high capacity while providing highest quality multimedia, versatile interfaces and all in a reliable, low-cost and small

format package. This means very advanced SoC designs and many overlapping projects in order to introduce new models at the rate that market leadership demands. Relentless and accelerated SoC project development demands reuse of FPGA-based prototyping methodology. We simply will not have time to re-invent wheels and methods or create large-scale prototyping hardware for every design. An in-house standard platform strategy and Design-for-Prototyping methodology will be required to keep all those software-dominated projects on schedule.

14.4. Prototyping future: networking

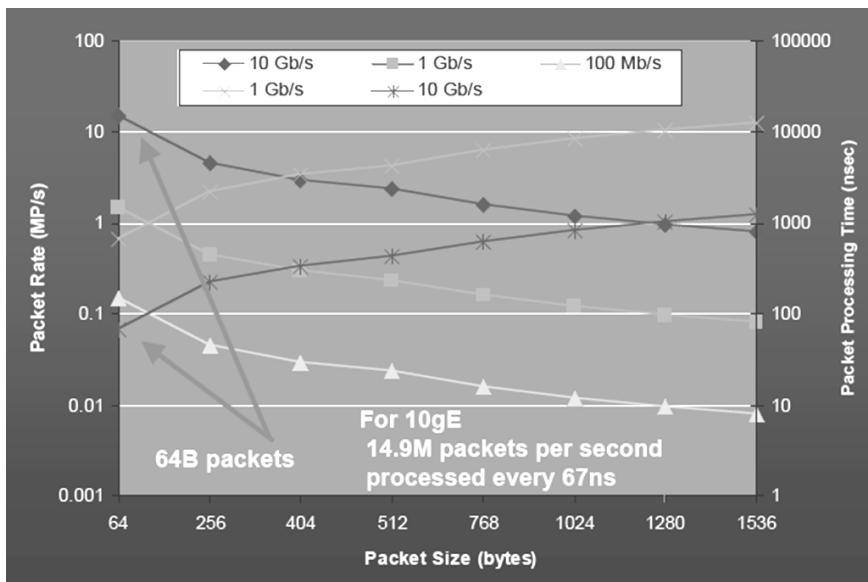
In contrast to the mobile wireless and consumer application spaces, networking is not determined by end-user applications but completely driven by data rates and per-packet processing requirements. Nevertheless, given the need for product flexibility and configuration options, software is once again used to determine processing functionality and the hardware is increasingly designed to prioritize the efficient execution of the software.

As already outlined in chapter 1, the networking application domain is moving towards architectures using multiple CPU cores and underlying flexible interconnect fabrics. As predicted by ITRS and shown in chapter 1, the die area will remain constant, the number of processors per design is predicted to grow by 1.4x every year, on average. This will have a profound impact on the future of prototyping given that the application partitioning between the processors has to be properly tested and verified prior to silicon production.

Considering multimedia applications briefly, a fair amount of the tasks to be distributed between processors can be pre-scheduled. Proper execution can be verified using virtual prototypes as they allow very efficient control of execution as well as the necessary debug visibility into hardware and software co-dependencies. For a networking application, however, incoming traffic is distributed to packet processors and on-chip accelerators. Given the inherently parallel nature of packet processing, the choice of compute resources is done at runtime, which means that less pre-scheduling is required. Nevertheless, the various options of runtime scheduling will need to be prototyped prior to committing to silicon, at as realistic speed as possible. FPGA-based prototypes and virtual prototypes will be enhanced to collect appropriate performance and debug data to optimize scheduling algorithms.

How the packet rates increase for different transfer rates and how packet processing times get shorter is illustrated in Figure 166.

Figure 166 : Packet rates and processing times (Source: AMCC)

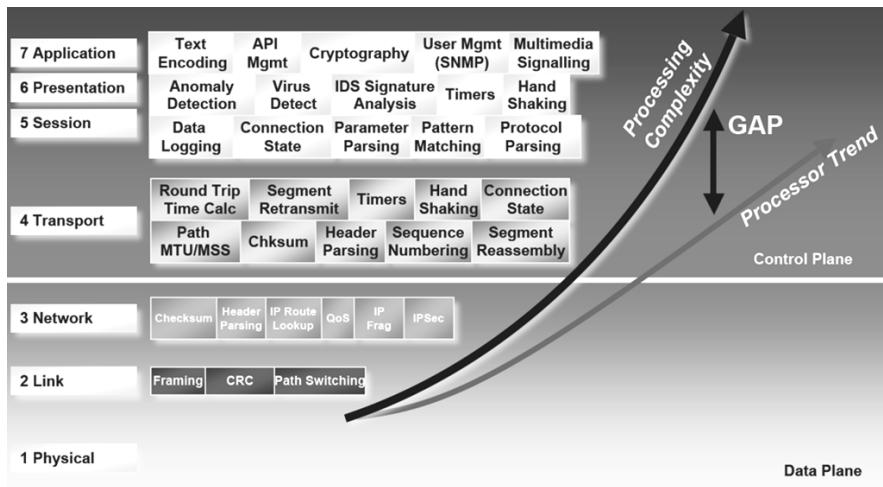


As an example for 10Gb Ethernet, 14.9 million packets have to be processed every 67 nanoseconds. To allow any kind of prototyping, both capacity and execution speed of prototypes will have to improve. For virtual prototypes it is likely that this can only be achieved using parallel, distributed simulation. FPGA prototypes will have to overcome capacity limitations using improved scheduling and partitioning algorithms as well as intelligent stacking of prototypes themselves.

Figure 167 illustrates how the different functions performed in the networking control and data plane have evolved over time as processors have become more capable. It also shows that the complexity of processing demands has been outpacing the development of new processors. Developers have therefore moved to multiple processor cores and the most complex challenge has become to efficiently partition tasks across them.

In the control plane, the “in-band control traffic” is handled with the actual connection processing for routing/session establishment of the network protocol in use.

Figure 167 : Functionality distribution in Data and Control Plane (Source: AMCC)



Taking into account the underlying multi-processing nature of the hardware, task distribution is important. Each “task” within the control processing is complex and offers limited internal parallelism. The tasks themselves are fairly independent and can be assigned individually to dedicated CPUs.

Unfortunately, network applications have processing loads that come in bursts, which leads to CPUs that are powerful enough to handle the peak load, but otherwise underused. As a result, these very capable processors are not an optimally efficient use of silicon. Given the dependency of the processing requirements on the actual networking traffic it is difficult, if not impossible, to assign tasks at compile time to different processors.

The data plane functionality is focused on forwarding packets and translates information from control traffic into device-specific data structures. The data plane's code is packet-processing code and easy to run on multiple cores operating in parallel. This code can be more easily partitioned across multiple cores, because packets can be processed in parallel as multiple cores run identical instances of the packet-processing datapath code.

Again, much like for control code, the actual assignment of processing units to tasks is highly dependent in the actual network traffic and should be done at runtime. Assignment of tasks to processors at compile time is, again, difficult if not impossible.

We predict that prototyping of networking systems will largely continue to be done in a hierarchical fashion. The individual processing units will continue to be tested against their packet-processing requirements but, given the increased complexity of

those requirements, it will become too risky to commit to hardware without proper prototyping.

As discussed earlier, in contrast to mobile wireless and consumer applications the assignment of processing tasks to processing units is not done at compile time and is also not separated as a set of user applications from the hardware through operating systems. The type of software used in networking applications is much more “bare metal” and tightly coupled to the processors upon which it runs. Hardware dependency in software is difficult to model without the hardware being present in some form. Hence software debug is posing different challenges, i.e., requires debug at runtime and is also driving requirements for redundancy, all of which will make prototyping of all types even more compelling than it is today already.

14.5. Prototyping future: automotive

Automotive is a very dynamic application area and particularly interesting in that it has complexity trends increasing in two different directions.

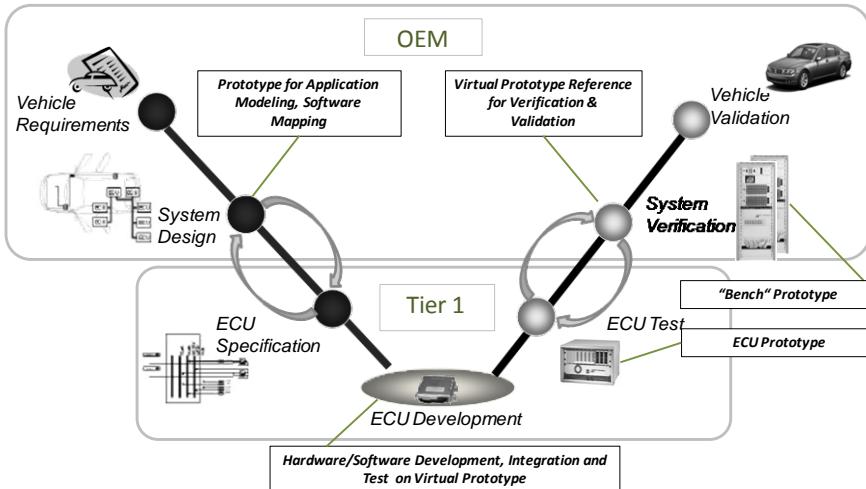
First, the overall complexity of a car as a combined device is already well exceeding that of wireless and consumer devices. Modern mid-range cars have at least 30 electrical/electronic systems with up to 100 microprocessors and well above 100 sensors. These processors will be combined into a network of so called engine control units (ECUs). With increased trends towards security and safety, as well as more and more video processing to assist driving, the number of ECUs and network complexity will grow much further and at an even faster pace.¹

Second, in automotive applications the complexity grows beyond just electronics, meaning the combination of electrical hardware and software. Automotive systems will also have to be developed which take into account mechanical effects and their interaction with electronics, so-called “mechatronics.” Clearly, any software in such a mechatronic system is very hardware-dependent indeed, once again increasing the need for prototyping.

As in many other application areas, software plays a crucial role so that hardware is increasingly developed with the aim of optimizing the software’s efficiency. Figure 168 shows the V-Cycle often used to represent the automotive design flow as a design starts from the left and progresses to the right.

The V-cycle diagram is a simple way to illustrate the interaction between the original equipment manufacturers (OEMs) who eventually produce the complete vehicle and the so-called tier 1 suppliers who design and deliver the ECUs for the various vehicle sub-systems. We can see on the right that a vehicle requirement is developed and is modeled at the system level until ECU-level specifications can be

Figure 168 : The automotive V-Cycle and its relation to prototyping



produced. Then the ECU supplier develops their product based on the specification and delivers units for test in the overall vehicle prototype for validation alongside all the other ECU being developed for the new vehicle.

Obviously communication and excellent project management is required between these two groups, and in the diagram above we can see two transitions at which communication between the various parties involved is crucial. However, growing complexity of electronic vehicle content is stressing traditional methods and it will be increasingly valuable for the communication between OEM and supplier to be more sophisticated in order to reinforce and illustrate the specification and deliverables.

Replacing paper specifications with executable virtual prototypes and FPGA-based prototypes will become mandatory to increase quality of communication and avoid costly turnarounds due to defects found to late in the design flow.

For example, if the specifications from OEMs to their ECU suppliers is imprecise then bugs are found during the integration phase, illustrated on the right side of the diagram, the development will have to go back all the way to the drawing board if defects cannot be corrected.

In addition, ECU developers need to be able to validate design trade-offs and efficiently interact with their suppliers and other ECU developers. Late changes in the specification can cause significant cost for correcting issues. The safety-critical nature of some of these vehicle functions is also demanding ever-more rigid and provable design and verification methodology, for software as well as hardware.

Figure 168 also illustrates the various stages at which prototyping would complement written or verbal communication with executable models of the systems, sub-systems and chips under development:

Before specifications are distributed to tier 1 suppliers, OEMs want to model the software applications and their effect on the system. Applications have to be modeled at high enough abstractions so that they can be mapped into early prototypes of hardware to be developed and effects like bus utilization, ECU and CPU utilization can be analyzed.

ECUs as sub-systems in themselves need to be prototyped to improve software development schedules by starting software development prior to hardware availability. The resulting prototypes (virtual, FPGA-based or using first silicon) can be used for hardware verification and system validation itself.

In the software world, once sub-systems are available and integrated, they are used for software testing as well. Hardware prototypes of the ECUs and sub-systems of ECUs are made available in complex test racks which are used for software development and testing.

Due to their networked nature, automotive systems are at least one order of magnitude more complex than, for example, consumer applications. Distributed simulation of virtual prototypes as well as increased capacity and scalability for FPGA prototypes will be key requirements to address the resulting capacity and speed issues. With the further increasing complexity and pressure on schedule timelines, more and more of these prototypes will become virtualized and already, hardware in the loop systems (HILS) are being used in conjunction with software simulation.

We can envisage completely virtualized systems in widespread use for high-level software validation, complementing FPGA-based or silicon-based prototypes which are located in main sites in smaller numbers, used for lower-level software validation.

Capacity, speed and early availability will be important selection criteria for the various prototyping options. Cost requirements will also drive further use of virtual prototypes for software development given that they have a lower cost point per seat than hardware-based prototypes, making them available in larger numbers.

In addition, safety requirements will change software development processes and make prototyping mandatory. Greater software content in vehicles means that

software validation becomes very widespread, beyond pure development into areas of certification and safety compliance. For instance, let's consider the development of the sub-system which angles the rear wheels depending on the vehicle speed around a curve. In the past it was based on pure mechanics and hydraulics and the safety of those components were well understood and could be verified by the appropriate testing authorities. Any attempt of replacing such a system with software on an ECU would require a great deal of software validation tests to be passed in order to pass safety certification, such as the emerging standards for software verification (MISRA) and predictable software development (ISO 26262). In such a validation-rich design environment as automotive, prototyping becomes a key factor as a means of early verification in repeatable design flows and an enabler for safety-critical software development.

14.6. Summary: software-driven hardware development

As outlined in the previous sections, the needs are different in the various application domains, but the overarching commonality is the trend towards more software development. In the future the actual system behavior in the majority application domains is mostly defined by software; that is certainly the case for those we explored in this chapter. As a result, hardware development will be driven to optimize the software's execution.

Depending on the specifics of the application domains this has different effects. In mobile wireless and consumer, the underlying digital hardware is fairly generic, enabling software platforms like Google Android™, Apple® Mobile OS or Windows Mobile™ 7 to execute most efficiently and decouple the ecosystem of application developers from the actual hardware.

In contrast, in the networking application domain, the software still defines system behavior but it is much more hardware-dependent, embedded in the networking software stacks on Linux and other operating systems. However, software is still the key differentiator of a combined hardware/software platform.

Similarly, in automotive, due to the overall complexity, developers of hardware-dependent software often reside in independent companies. They need access to representations of the hardware for which they are developing software and this increases the need for virtual and FPGA-based prototypes.

There is no question that software will gain even more importance and that we are on cusp of quite significant changes in development methodologies as well as responsibilities within the design chains. How exactly the changes will manifest themselves, very much depends on the application domains. It is safe to assume that in all of them prototyping will play a key role.

14.7. Future semiconductor trends

Having considered the future from the viewpoint of certain key application areas, let's cross-reference those predictions by taking a look at the trends in semiconductor development and production.

In chapter one we outlined the various semiconductor trends that have impacted SoC and other chip design up until now. We saw how these trends have increased the need for prototyping up until the present time. In this section we will complete that task and try to predict how those trends will continue over the next five years and what effect this will have on prototyping.

- **Miniaturization:** there is no clear end in sight to the miniaturization of silicon to achieve smaller silicon technology nodes. As a result, the complexity of the projects at the leading-edge technology nodes will be so complex that it is simply too risky to tape out without prototyping early and often. FPGA-based prototyping will support increased complexity as FPGA devices get larger, benefiting, and to some extent, driving those technology trends.
- **Embedded CPUs:** as the latest generations of FPGAs have family members which include embedded processors, it will be interesting to see if they will be used to run the embedded software in the system, rather than use a test chip of the CPU core(s) which the FPGA is prototyping. Perhaps the choice of CPU in the SoC might even be driven by its availability or otherwise in an FPGA format for prototyping.
- **Decrease in overall design starts:** this trend is widely expected to continue as the SoC production costs will make smaller technology nodes less accessible and will likely cause further consolidation in the semiconductor industry. With less design starts the remaining designs need to address more designs in order to re-coup the investment through more end applications. Virtual and FPGA-based prototyping will become even more necessary in order to mitigate risk of potential re-spins.
- **Programmability:** in the mobile wireless and consumer application domains the desire to de-couple software development from hardware dependencies will further increase. Virtual prototyping will also gain in importance as it allows software development to commence even earlier. SDKs will contain greater capabilities for software verification previously only accessible to host development.
- **IP reuse:** IP usage continues to increase, which is an easy prediction to make. The semiconductor analyst Gartner confirmed its most recent predictions that the amount of IP reuse will again double between 2010 and

2014. In addition, the trend to licensing complete sub-systems will grow and open a new area of prototyping for complete sub-systems containing an assembly of pre-defined hardware and software IP.

- **Multicore processing:** Adoption of multicore architectures will cause more pressure on analyzing and optimizing software parallelization. Today parallelization has been solved in specific application domains, such as graphics, but it is likely that different application-specific solutions will be required in other areas, such as networking and automotive electronics.
- **Low power:** today's methods for reducing power in semiconductors are focused on implementation and silicon-level engineering. Future requirements will be better addressed by moving the focus of low-power design to the architectural design level. As a result virtual and FPGA-based prototypes will be instrumented to allow low-power analysis for early feedback on some aspects of the power design. For example, for activity capture and average dissipation over certain software functions.
- **AMS design:** an increase in the analog/mixed signal portion of chips will create even more demand to allow in-system validation. Virtual IO for virtual platforms and interfaces of FPGA-based prototypes to its environment will become more critical.

14.8. The FPGA's future as a prototyping platform

As well as the previously mentioned trends which demand greater prototype adoption, there are trends in device and tool capability which allow us to keep up with that demand. FPGA devices are a classic illustration of Moore's Law, as we mentioned before. New research will allow the use of multi-die packages and 3D technology to greatly increase the capacity of our leading-edge FPGA devices beyond Moore's Law. However, this leap in capacity brings new challenges in connectivity. The ratio between internal resource and external IO will continue to cause visibility and connectivity issues for prototyping, accelerating novel solutions for multiplexing and debug tools.

The complexity of creating reliable and flexible FPGA boards using these new devices in ever-shorter project cycles will reduce the proportion of in-house boards compared to commercial boards. In-house boards will still be used for specific needs or to support very large numbers of platforms, or to support a large company in-house standard. However, we shall see most other designs complete their current migration to ready-made boards, which itself will probably create a healthy and competitive market from which prototypers can select.

The future of FPGAs as the hardware prototyping platform of choice is secure and will be complemented with growing use of virtual prototypes. These will

increasingly be combined into hybrid arrangements and we can expect a merging of the two approaches into a more continuous prototyping methodology from system to silicon in the future.

14.9. Summary

Given the analysis of the previous sections in this chapter, prototyping will become even more of a key element for future design flows. Interesting times are in store for providers of prototypes at all levels. Eventually both the hardware and software development worlds will grow closer together with prototypes – virtual, FPGA based and hybrid – being the binding element between both disciplines.

The authors gratefully acknowledge significant contributions to this chapter from:

Frank Schirrmeister of Synopsys, Mountain View

CHAPTER 15

CONCLUSIONS

We have come to that point that occurs in every book; the end. With a technical book, such as the FPMM, we also need to summarize the lessons learned in the book and reach some conclusions. In this very short chapter we shall endeavor to capture the essence of the previous 400 pages in a short take-away form.

15.1. The FPMM approach to FPGA-based prototyping

Table 35 gives a summary of the various steps and main considerations in running an FPGA-based prototyping project.

Table 35: Summary of steps in FPGA project

Start from the beginning	Get involved at the earliest stage of the SoC project; not after lots of FPGA-hostility has already been introduced into the design.
Understand the goal	How much to prototype? What speed? How many platforms? How many versions? What deadlines?
Choose platform	Speed and capacity are important but flexibility is critical. Maintain high quality and reliability.
Modify RTL	Remove FPGA-hostility and add necessary items for FPGA operation. Keep revision control.
Partition	Aim at balancing resource usage and minimizing number of interconnect signals.
Reconnect	Fix IO and multiplex between FPGAs as required. Use high-speed differential signaling as required.
Bring-up	Incrementally introduce design onto boards while checking for signs of life and correct operation.
Debug	Start using the prototype to debug the design and the software which runs upon it. Have a debug plan. Use incremental tool flows.
Replicate and use	Make enough copies for every end-user and support these to success.

We have addressed each of these steps throughout the chapters of this book and taken one step at a time, and using external pre-made sources as often as possible, we can quickly gather the expertise and tools required to create a successful prototype. Let's reconsider our three laws of prototyping as reprised in Table 36.

Table 36: The three “laws” of prototyping revisited

Law 1:	SoCs are larger than FPGAs
Law 2:	SoCs are faster than FPGAs
Law 3:	SoC designs are FPGA-hostile

Considering each law in turn, we can summarize the main FPMM conclusions that we found in each case.

15.2. SoCs are larger than FPGAs

Despite, or perhaps because of, 25 years of progress, the first law of prototyping is still pertinent. It is true that many FPGA-based prototyping projects require only one FPGA and this is usually true for IP block verification and in-lab feasibility projects. However, SoC projects are by their nature large and getting larger and while FPGA devices are at the leading edge of silicon development and also getting larger year-on-year, we will probably still need multiple of the largest FPGA devices to prototype a full SoC.

This means partitioning of the SoC design into multiple FPGAs for which we recommend using specialist EDA tools. These will allow us to partition interactively or automatically, the former giving the better results but taking longer.

The typical size of a prototype is between one and ten devices but with sophisticated automated partitioning and programmable interconnectivity, prototypes up to 20 devices are possible. As each device is already just over four million ASIC gates, this represents a design up to 80 or 90 million SoC gates which is way above the size of the majority of SoC designs today.

When the SoC design is partitioned over multiple FPGAs, we will need to perform some extra design work in order to maintain the design connectivity, the clock networks, and the reset and start-up synchronization. These are all explained in chapter 8.

15.3. SoCs are faster than FPGAs

The second law is true when comparing the core speed of both types of device, where the fine-grained and infinitely versatile architecture of an SoC customer chip means that it can outpace an FPGA, with its coarse-grained programmable architecture. So we should expect the RTL from an SoC design with a core speed of many hundreds of megahertz to run in an FPGA with core speeds in the region of 20MHz to 100MHz.

However, this speed is reduced if the design's critical path crosses the FPGA boundary and especially if multiplexing is used between the FPGAs. Both of these situations are common so the typical performance of a prototype in most cases is between 10MHz and 25MHz. Nevertheless, this prototype speed represents by far the fastest pre-silicon platform possible for exercising the RTL of an SoC design.

So the core speed of an FPGA is relatively fast but the first law holds true. At the IO the FPGA is just as fast as the SoC silicon in most regards and is able to support very much the same standardized peripheral interfaces as the SoC. Indeed, the ability of the FPGA IO to reproduce fast peripheral interfaces is one of their most valuable advantages over slower verification technologies. In addition we can use external PHY devices as well as those built into the FPGA.

All this means that it is common to find FPGA-based prototypes USB at 125MHz, or HD video running at full 72MHz rates. For the IO, therefore, with care and the right IP, we can break the second law of prototyping even if it still holds true for the core logic.

15.4. SoCs designs are FPGA-hostile

After reading chapter 7 we might be forgiven to think that FPGA-based prototyping involves quite a few changes to the SoC RTL but that does not always need to be the case. That chapter covered many possible scenarios and it would be a pathological design indeed that needs all of them to be fixed but, nevertheless, a good prototyping team will not be daunted by even the most FPGA-hostile SoC design. Armed with the best tools, IP, boards and devices and most importantly the right expertise and approach, we can tackle any SoC design and create an FPGA-ready version of at least the majority subset of the design.

However, that is only the minimum acceptable result of a FPGA-based prototyping project and far more is achievable. As we outlined in chapter 9, a Design-for-Prototyping approach will ensure that the SoC design does not arrive as FPGA-hostile in the first place.

We do not have to accept the third law of prototyping as a simple fact of life. A Design-for-Prototyping approach means that we can break the third law and make our SoC designs, if not FPGA-friendly, then at least FPGA-tolerant!

15.5. Design-for-Prototyping beats the three laws

Design-for-Prototyping guidelines fall into two groups; procedural guidelines and design guidelines.

We summarize the procedural guidelines in Table 37 and the design guidelines are not repeated here but instead are included in full in chapter 9.

It is hard to imagine that we will be fully successful if we only follow one set of the

Table 37: Summary of procedural recommendations in Design-for-Prototyping

Recommendation	Comment
Integrate RTL team and prototypers.	Same tool training. Shared knowledge of FPGA and SoC
Prototypers must work closely with software team.	Software team are the prototypers best “customer”
Include prototype in the verification plan.	Branch RTL at pre-agreed milestones of maturity
Keep prototype-compatible simulation environment.	To compare results with original SoC simulations
Keep combined documentation and revision control.	Track software and RTL changes for prototype
Adopt company-wide standard for hardware and add-ons.	Avoids waste and encourages re-use
Include Design-for-Prototyping in RTL coding standards.	Design-for-Prototyping RTL style is also good for SoC.

guidelines, however. If we had to choose just one, then it is the procedural guidelines that are probably the more important set.

The secret of success in many prototyping projects is in this integration of the prototypers with the rest of the SoC design team. This must lead to the inclusion of the prototypers' needs and their feedback at the early stages of the SoC project. Then, with only small changes to procedure and design style, the success of FPGA-

based prototyping can be guaranteed. In return, the software and RTL parts of the SoC are introduced at the earliest possible opportunity with minimum effort, addressing the critical development effort of most SoC projects today, the embedded software.

15.6. So, what did we learn?

The main lesson might be to tackle complexity one step at a time but make sure that each step is in the right direction and not into a dead end. When starting an FPGA-based prototyping project, our success will come from a combination of preparation and effort; the more we have of the former, the less we should need of the latter.

Having read through most of this book, we will enjoy a significant head start to our prototyping projects. We hope that our methodical approach to FPGA-based prototyping will allow anybody to tackle the complexity of their next SoC project with more confidence and achieve even more successful results.

APPENDIX A.

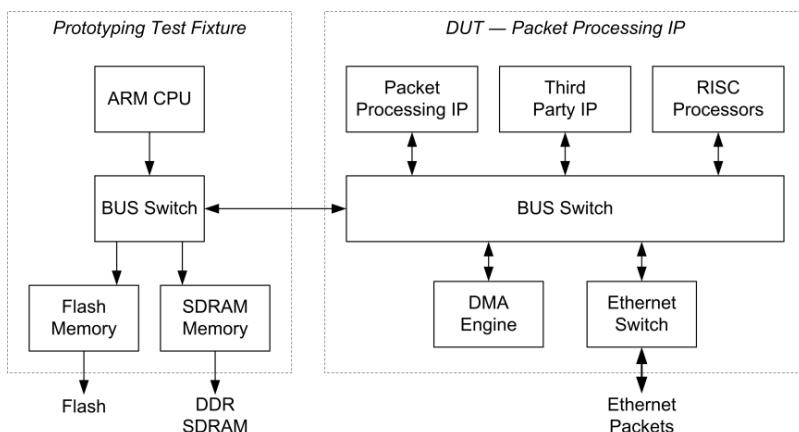
WORKED EXAMPLE: TEXAS INSTRUMENTS

It is always instructive to learn from those who have gone before us. This appendix provides details of an FPGA-based prototyping project performed by a team in the Dallas DSP Systems division of Texas Instruments under the management of David Stoller. David is an applications engineer who has completed the RTL and systems verification of numerous IP modules and sub-systems integrated in cable and DSL modems, IP phones, and other networking-based products from Texas Instruments.

A1. Design background: packet processing sub-system

This case study examines the FPGA prototyping efforts within Texas Instruments of a complex packet processing sub-system as seen in Figure 169.

Figure 169: Packet processing sub-system block diagram (source: Texas Instruments)



This particular sub-system consists of the following IP subcomponents each of which are tested individually or as part of the complete sub-system:

- DMA engine and packet-processing sub components
- Third-party IP
- 3-port gigabit Ethernet switch
- Firmware-based RISC processors
- Proprietary bus infrastructure

Within Texas Instruments, a typical IP design and verification cycle consists of the following steps:

1. The IP design team defines the feature set of the IP to meet the application requirements, then specifies the IP and implements the design in RTL.
2. The IP design team performs basic functional “smoke” testing to ensure that the RTL exhibits basic functionality.
3. The IP design verification (DV) team then performs comprehensive simulation verification by implementing and running a large suite of test cases. When the IP is passing approximately 80% of the planned test cases, the design is considered robust enough for FPGA implementation.
4. The IP prototyping team integrates the IP RTL into FPGA RTL, compiles the design for the targeted FPGA platform, and brings up the platform to begin FPGA testing.
5. When the simulation and FPGA verification are complete, the IP module is released to the chip team, where it is integrated into the chip RTL.
6. Chip-level verification is performed on the entire chip SoC through both SoC simulation verification and the use of a commercial emulator platform.

A2. Why does Texas Instruments do prototyping?

Each and every step in the design and verification flow is critical to the success of an IP block but the goal of the FPGA-based prototyping step is to perform pre-silicon verification of a complex IP block. FPGA-based prototyping allows the prototyping team to target conditions that are not easily replicated by alternate verification methods, including:

- Higher frequency RTL clocking, yielding very large numbers of test cycles.
- Memory transactions with real memory devices and other external peripherals.
- Networking protocol checking with actual networking equipment (USB, Ethernet, PCIe, etc.)
- Simultaneous interactions between multiple masters and peripherals or memory controller.
- Complete sub-system running at scaled clock frequency with heavy loading of interconnect buses to determine maximum throughput of logic and interfaces.
- Channel setup and teardown operations with and without traffic present.
- Testing IP running with actual asynchronous clock boundaries.
- Running real software applications which involve actual CPU instruction and data cache operations as well as management of peripherals through interrupt servicing.

Owing to the sheer number of gates present in the complete chip, the prototyping team did not intend for the FPGA prototyping platform to encompass the entire SoC. Instead, the primary goal was to map a part of the SoC that spanned 1-4 of the largest available FPGAs and could be run at approximately 10MHz. This allowed for a reasonable compromise between size and frequency to allow for useful FPGA prototype testing.

A3. Testing the design using an FPGA-based prototype

The IP DUT was tested by integrating the DUT RTL into an ARM® processor based sub-system which was then mapped onto the FPGA platform. This was previously illustrated in Figure 169.

The ARM processor served as the test host for the FPGA prototyping verification. Test applications written in “C” were run on the ARM to exercise the DUT with various scenarios that were defined in the FPGA test plan. In most cases, the memory map and infrastructure of the ARM sub-system were matched to that of the targeted chip. This allowed the FPGA platform to be used for software application and driver development before silicon was available. Additionally, having the FPGA test sub-system architecture match or resemble the architecture of the actual chip allowed for more meaningful performance measurements on the FPGA DUT.

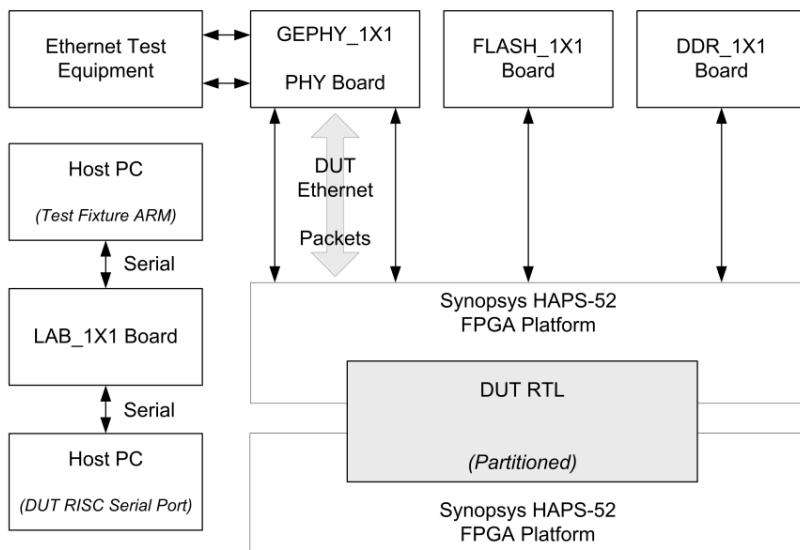
The packet processing testing scenarios were broken up into two major classes.

Loopback tests were implemented by the ARM test application generating packets, sending them into the packet processing sub-system, receiving packets back from the hardware, and then checking the packets for data integrity. The packet loopback was performed at the boundary of the system (in the Ethernet PHY). Loopback tests were specifically designed for testing various packet configurations and extensive checking of all received packets through the ARM software.

Echo tests were implemented by sending packets from the Ethernet test equipment into the packet processing sub-system. The DUT routed the packets through its various hardware blocks, and then the packets were transmitted back to the Ethernet test equipment where they were checked for errors. Echo tests were specifically designed to maximize the packet throughput through the DUT, exercising the various parallel data paths, and provide minimal checking of errors among received packets.

Figure 170 provides an overview of the FPGA prototyping platform that was assembled for this project.

Figure 170: FPGA prototyping hardware used in the lab (source: Texas Instruments)



In this example, the DUT ARM processor executed a simple bootloader/OS out of flash memory in the same manner that the processor will on the final silicon. Test

applications were loaded onto the flash or DDR SDRAM daughter board using either the serial port or ARM JTAG debugger on the LAB_1x1 board. The ARM executed the test application out of SDRAM and configured the DUT in preparation for running the tests. Ethernet packet streams were configured on the Ethernet packet generator which sourced them to the hardware platform on 10/100/1Gb interfaces to be processed by the hardware in echo tests. The Ethernet test equipment also served as a packet “checker” and verified that all expected packets were received, had no errors, and maintained the expected throughput rate.

A4. Implementation details

The entire packet processing RTL spanned four Xilinx® Virtex®-5 LX330 FPGAs (on two HAPS®-52 platforms or a single HAPS-54 platform). The final partitioning consisted of the following FPGA utilizations:

FPGA A 60% LUTs used
FPGA B 92% LUTs used
FPGA C 83% LUTs used
FPGA D 20% LUTs used

The FPGA implementation of the DUT was configured for two different clocking scenarios; “high-speed” and “low-speed” scenario.

The initial clock “high-speed” frequency targets were:

Packet Processing IP:	37.0MHz
Ethernet IP:	12.5MHz
DDR SDRAM:	50MHz

Following the partition across 4 FPGAs using pin multiplexing techniques, the operating frequencies were lowered as follows in order to overcome some clocking limitations:

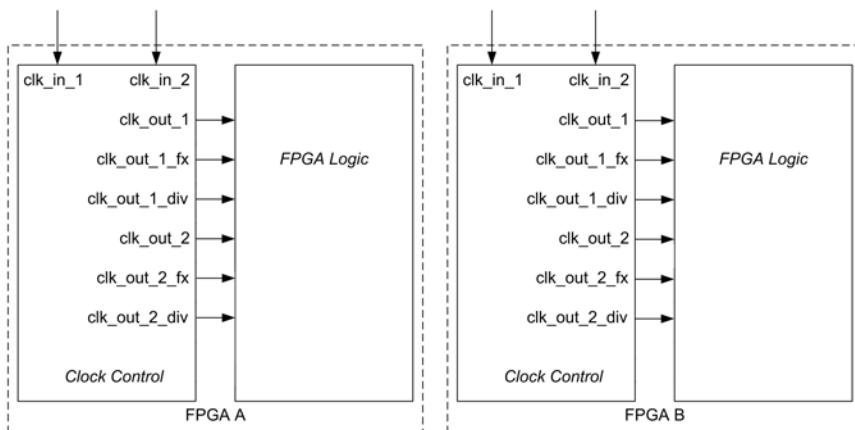
Packet Processing IP:	7.0MHz
Ethernet IP:	12.5MHz
DDR SDRAM:	50MHz

Both scenarios yielded useful results and each shall be explained in detail, starting with the high-speed scenario.

A5. High-speed scenario

In order to properly test the packet throughput and pushback, specifically at the boundary between the 3-port gigabit Ethernet switch and the rest of the packet processing sub-system, it was necessary to set up the DUT to run at 1/10 of the clock speeds of the actual chip. By allowing the silicon clock ratio to be maintained on the FPGA platform, it ensured that each of the DUT components see the same level of transaction activity as the actual chip. For a silicon chip operating with a gigabit stream of Ethernet traffic running at 125MHz and the packet processing sub-system running at 370MHz, the corresponding FPGA-prototyped sub-system operated with a 100Mbit stream of Ethernet traffic running at 12.5MHz and the packet processing sub-system running at 37MHz.

Figure 171: Clocking structure for high-speed scenario (source: Texas Instruments)



While this scenario satisfied the particular goal of testing the asynchronous boundary, it did present several challenges, including: FPGA interconnect pin limitations, FPGA capacity limitations, and the FPGA DCM minimum clock frequency. Both the interconnect and capacity limitations of the hardware platform drove a need for the platform to be implemented at these clock frequencies with a reduced DUT sub-system with some modules removed.

The clocks in this high-speed scenario were generated from a number of DCMs placed in the clock control block in Figure 171. However, by using the Virtex-5 DCMs to generate all of the required system clocks, the platform was limited to the combinations of clock division and phase shifting that could be achieved due to the 32MHz minimum clock input to the DCM.

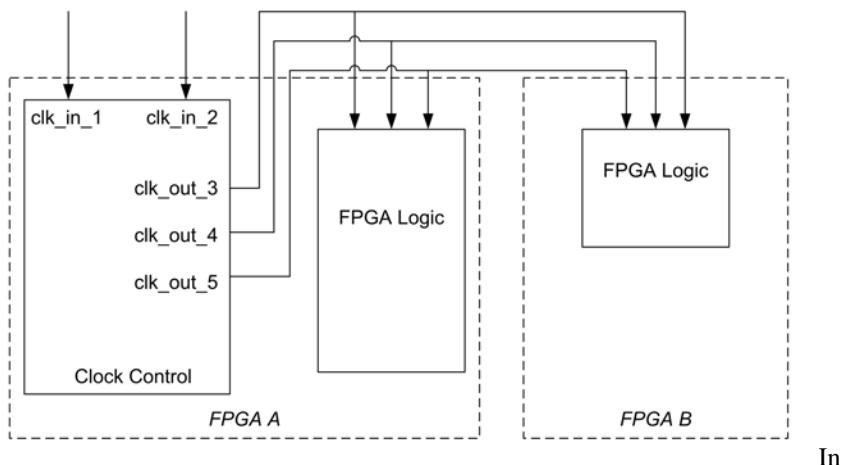
This factor created issues with generating divided down or phase-shifted clocks which prevented certain IP modules from being included in this phase. Additionally,

divided down clocks from the DCMs could not easily be used on multiple FPGAs, as it was difficult to ensure that they are synchronous and in phase.

A6. Low-speed scenario

The low-speed configuration was implemented in order to address the FPGA pin interconnect and capacity limitations of the hardware platform. By eliminating the requirement of implementing a packet processing IP/Ethernet IP clock ratio of 37/12.5, the FPGA RTL was reconfigured to include all of the DUT IP components (transitioning from two to four FPGAs) and to allow all of the resulting IP interconnect to be routed using the Synopsys® HSTDm pin multiplexing scheme.

Figure 172: Clocking structure for low-speed scenario (source: Texas Instruments)



In order to resolve the limitations presented by the original clocking structure in the high-speed scenario shown above, the clocking structure in Figure 172 was implemented.

The low-speed clock scenario consisted of a central clock control module instantiated on one of the four FPGAs that generated all necessary system clocks using a simple FF counter clock divider scheme. This allowed the ability to generate unlimited divided and phase-shifted clocks from a single clock input fed through a DCM. These generated clocks were then fed out of the FPGA's clock outputs, where they were routed back into each of the FPGAs on the platform.

This clocking structure had several advantages. First of all, this scheme allowed for a lower operating clock frequency that did not rely on the input requirements of the FPGA DCM. This low speed was necessary to implement pin multiplexing (HSTDm) between the FPGAs and therefore overcome any interconnect limitations. The low clock frequency also significantly reduced the FPGA synthesis and place & route runtimes. By having all of the generating clocks exit the FPGA outputs and then re-enter each of the FPGAs on their clock input pins, the FPGA compile tools could effectively ignore any of the clock generation circuitry in their analysis. This clock structure also provided much more flexibility when prototyping an IP that required a set of divided or phase-shifted synchronous clocks.

A7. Interesting challenges

The biggest challenges faced during the FPGA prototyping testing of the packet processing sub-system involved overcoming limitations of both FPGA capacity and FPGA pin interconnect.

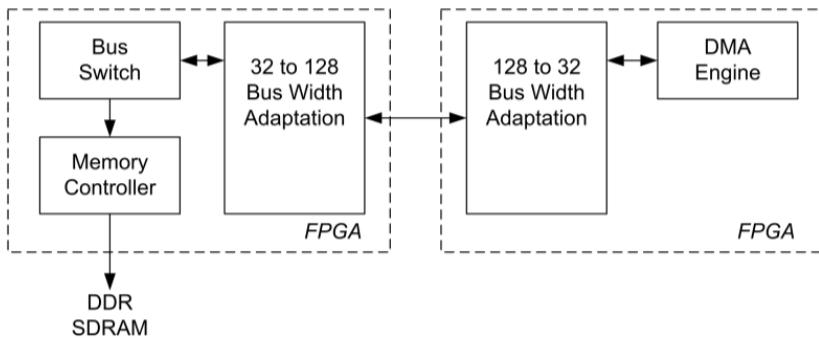
FPGA capacity limitations were addressed by proper partitioning of the DUT RTL using manual methods as well as the Synopsys Certify® tool. This resulted in spreading out the DUT across two or four FPGAs. When done manually, this was a labor-intensive step which involved attempting various partitioning scenarios until the partition is successful. Using Certify eased much of the overhead involved in performing the complex partitioning.

FPGA capacity limitations were also addressed by constructing several configurations of the DUT RTL in which various sub-modules were removed in order to reduce the overall size of the sub-system. This was successful particularly when the IP contained several distinct functions, each of which could be separately tested. In some cases this allowed the DUT (with reduced resources) to be mapped across two FPGAs, allowing more efficient usage of multiple FPGA platforms by the team. However, this solution was entirely manual and added significantly to the time- and effort-resources spent on the FPGA implementation portion of the testing project.

When the packet processing RTL was partitioned across multiple FPGAs, there was often a conflict between how many pins were available between the FPGAs and how many were required by the internal signals within the DUT. Additionally, there was a trade-off between using the FPGA pins for external daughterboard connections and reserving the pins to accommodate FPGA-to-FPGA interconnect.

Initially, pin interconnect limitations were addressed by adding bus-width adapting bridges to internal buses within the DUT (i.e. 128 bit adapted down to 32 bit). This solution shown in Figure 173, allowed for the FPGA operating frequencies of the sub-system to remain in the high-speed clocking scenario, although with a side effect, as it did modify the architecture (and possibly the functionality) of the DUT.

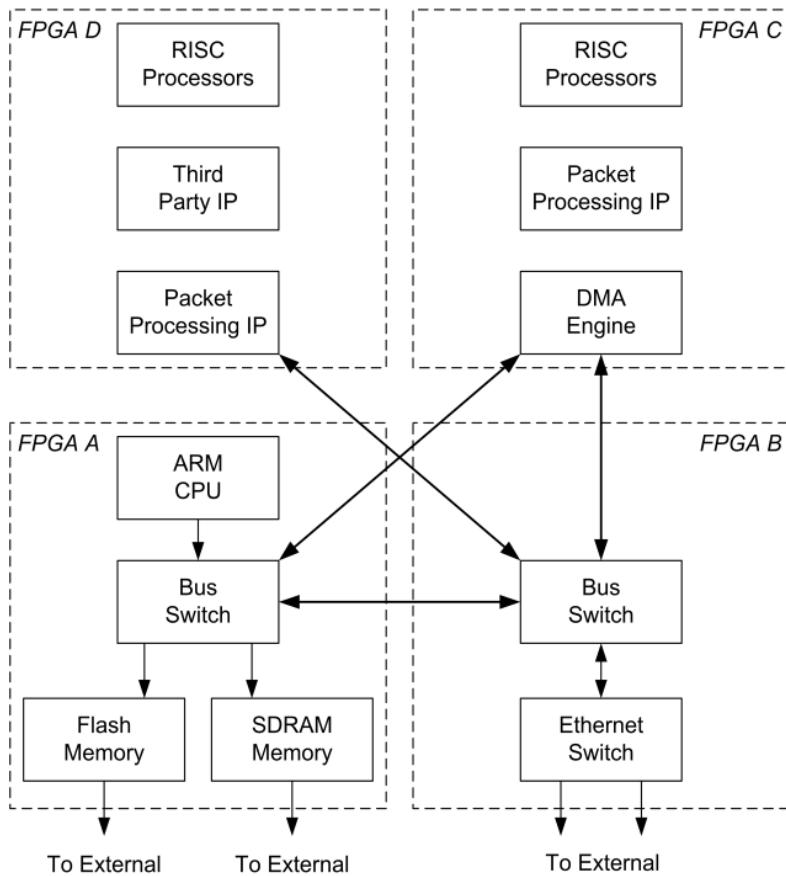
Figure 173: Block diagram of bus-width adaption (source: Texas Instruments)



In order to implement and test the DUT without the presence of the bus width adaption bridges, the HSTDm pin multiplexing scheme within the Synopsys Certify tool was utilized. The HSTDm methodology was implemented using a 16:1 multiplexing ratio which allowed for all of the internal nets to be connected when the entire packet processing IP was partitioned across four FPGAs, as shown in Figure 174.

The trade-off with this methodology was that the HSTDm required a lower operating frequency and this was the overall driving factor in switching to the low-speed clocking scenario.

Figure 174: Block diagram of partitioned sub-system (source: Texas Instruments)



Summary of results

The FPGA-based prototyping team at Texas Instruments was able to construct a complete configuration of the packet processing sub-system and fully load it with multiple Ethernet streams to prove that the throughput and latency met the chip specifications. Once the DUT was up and running on the FPGA platform, hundreds of tests were run, uncovering multiple issues including several system-architecture problems that were impossible to replicate on a simulation or emulation platforms.

The FPGA platform was also used as a software development station for multiple engineers to perform software driver testing and firmware-based testing and development. The platform was operated in a time-sharing mode between the local hardware verification engineers and the remote software-verification engineers. Software engineers at a remote site were able to remote login to the host PC, which would allow them to access the ARM and RISC processor debuggers, to reset the platform when needed, and to perform their testing. They were able to load system-level tests and drivers onto the ARM processor and verify software applications prior to silicon being available.

Additionally, the software engineers were able to verify the firmware targeted for the various RISC processors in the packet-processing IP and demonstrate full functionality prior to silicon being available.

In addition to its hardware-verification advantages, the FPGA-based prototyping platform was demonstrated as a much more efficient software development platform than the emulator, providing a lower-cost solution, delivering many more test cycles and also having the ability to interface with a software debugger and real-time Ethernet test equipment.

APPENDIX B. ECONOMICS OF MAKING PROTOTYPE BOARDS

In chapter 5 and chapter 6 of this manual we explored the details of both making and buying a suitable FPGA platform. We tried to avoid making any value judgments between these two approaches because in any given situation one or the other may be most appropriate. This appendix is provided as a guide to help readers examine the business reasons for choosing between “make” and “buy.”

B1. Prototyping hardware: should we make or buy?

A wrong decision regarding the exact FPGA platform may have great impact on the cost and overall success of the whole prototyping project. The relative costs of various options might be the overriding consideration, but it is essential that the whole argument is considered. After all, if we have to compromise our prototyping goals to meet a budget, then let us at least be sure we understand all the cost factors involved.

The typical questions that arise when considering the best way to implement an FPGA prototype include “*do I have enough time or expertise to create and manufacture a fully custom board?*” or “*will commercially-available boards cause me to compromise my goals too much?*”

This section aims to expose the most significant factors to consider when choosing to develop or purchase prototyping hardware. Costs, development time, and effort are considered in detail; including the obvious direct factors of bill-of-materials (BoM) cost, manufacturing time and test yield as well as some related business issues, such as engineering opportunity cost, and return on investment (ROI).

Some time ago, the authors created a cost comparison spreadsheet (CCS) which is helpful in arranging all the cost and time factors involved. In the hands of a project manager in position of the ambient facts, the CCS allows us to explore the arguments based on our own situation. The references at the end of this book give details on how to obtain a copy of the CCS.

B2. Cost: what is the total cost of a prototyping board?

An FPGA-based prototype board is worth considerably more than the sum of its components. Comparison should not be made by simply comparing BoM cost of an in-house board vs. the purchase price of a finished commercial board. Many other costs beyond BoM should be amortized across the in-house board before cost comparison is made realistic. These costs can be grouped into direct and related business costs as shown in Figure 175.

The following sections discuss each of these costs in depth.

Figure 175: Main costs associated with custom board development

<i>Direct costs</i>	<i>Related business costs</i>
• Personnel	• Time
• Equipment and expertise	• Risk
• Material and components	• Opportunity
• Yield and wastage	
• Support and documentation	

B3. Direct cost: personnel

Design setup cost: prototyping requires specific FPGA expertise to perform set-up tasks which are independent of the target board, as explained in depth in chapter 7. Engineering effort will be expended on this kind of design manipulation regardless of our choice of board. A typical project might take four to six weeks to be set up by such experts. Using our CCS and entering typical data for engineering costs and overheads, we find that this set-up cost could typically be \$20,000 per project (apologies to those reading in other countries but all cost examples in this appendix will be in \$). For novice teams, the set-up time would be longer and costs would be proportionately higher.

Board engineering cost: there is an obvious difference in engineering cost between developing in-house boards and buying them ready-made. In the former case, the cost of development, manufacture and test must be borne by the project. In the latter case, it is the capital cost of buying in the board(s) that must be borne. Managers will understand that there is a difference between the visibility of project costs and visibility of capital costs at most corporations. Whereas the capital cost of purchasing ready-made boards is fairly obvious, the engineering cost of in-house

development must also be fully understood in order to make an accurate comparison.

We must therefore have a figure for the development cost for the boards and then amortize this across the total number of boards in order to reach a per-unit development cost. To calculate a per-unit cost we should consider all relevant engineering costs, including:

- Development engineering
- PCB layout
- Manufacture engineering
- Test engineering

Once again, readers are urged to make calculations using their own figures; you will need to enter your company standard overhead and opportunity cost multipliers (e.g., overhead adds 50% cost on top of salary), and also estimate the time taken by each of the engineers to complete their tasks (e.g., PCB layout might take 40 days). The model used in the CCS includes these factors.

For example, 40 days is not unreasonable for a complex four-FPGA board with large BGA packages, trace-length matching, and 25+ layers. Using typical information entered into the CCS, cost for a 40-day PCB layout task may be estimated as \$40,000, which is then amortized across the quantity of boards that are built and used in the project.

B4. Direct cost: equipment and expertise

General equipment is accounted for in the overhead multiplier on the personnel calculation above. However, in addition, there will be specific capital equipment employed to perform some of the design tasks. For example, a PCB layout station, manufacturing equipment, or test equipment. If there are production projects queuing up for the equipment while it is being used to create prototype boards, then there will be an opportunity cost associated with that. It is difficult to quantify that material opportunity cost and so they are not included in the CCS but you should consider where potential bottlenecks may be in your own capacity to design, build, and test boards.

Figure 176: Engineering costs as entered into CCS

Overhead per engineer (% of salary)	50%
Annual Salary for circuit designer	\$80,000
Annual Salary for PCB designer	\$70,000
Annual Salary for test engineer	\$60,000
Annual Salary of FPGA Expert	\$80,000
ROI in design personnel and equipment	125%
Cost of Board Assembly Station per day	\$2,000

In the CCS, you can choose to factor equipment opportunity cost into the opportunity cost of the design personnel themselves. In any case, equipment opportunity cost should not be ignored. The CCS does, however, differentiate between in-house assembly costs and outsourced costs, including any non-recurring engineering (NRE) costs.

Engineering cost example

Figure 176 shows how typical data can be entered into the CCS. In this case we are making broad assumptions for salaries, overheads, and ROI.

To clarify, the overhead of 50% means that for each dollar paid to the engineer, there is another 50 cents of extra cost (e.g., equipment, employment taxes, benefits etc.). The ROI figure attempts to model the opportunity cost; a figure of 125% means that the company would normally expect a return of 125 dollars of revenue for every 100 dollars paid in salary plus overhead for these engineers.

These figures are used for the calculations and for the purposes of this example are deliberately underestimating the real engineering costs. At risk of being repetitive, you may work with the CCS to enter your own information.

The engineering cost also depends upon how long these engineers are employed on the board development. We therefore have a section in the CCS in which we enter the estimated time required for the development.

Estimates for time are difficult so we recommend using real data from a recent prototyping project rather than estimates for a future project. The CCS example in Figure 177 shows estimates based on real board designs known to Synopsys®. As is good practice for project management, there is a spread of estimates for each figure, which is given in resource-days. The CCS takes into account elsewhere how many engineers in each discipline are available for the project.

Figure 177: Estimates for engineering effort entered into CCS

	Board Engineering Time	
	Min	Max
Circuit design (resource-days)	20	40
PCB layout (resource-days)	40	50
PCB Manufacture (days)	10	20
Assembly Set-up time (days)	3	5
Assembly time per board (days)	0.5	0.5
Test time per board (days)	1.0	2.0

We will see later how these figures impact the results of the CCS.

B5. Direct cost: material and components

Now that the board has been designed, it has to be built and this requires BoM management and component purchase effort. Simplistic analysis of “make vs. buy” decisions might only compare the component cost of the in-house board with the purchase price of the off-the-shelf board. As we have already seen, this ignores many significant costs.

Your own purchasing professionals will be very efficient in managing BoM costs for your own products in volume. This efficiency may not be fully transferable to very low volume runs such as prototyping boards. The extra effort to purchase and manage the BoM for a few FPGA-based prototype builds may not be a good use of their time or your component management system. The numbers are hard to measure but it is estimated by volume manufacturers that annual costs for maintaining items for production may be over \$500 per different component type. When we consider that a four-FPGA board may have over 100 different component types, we can see that this becomes significant. This annual cost would be relevant if you are planning to use the boards in more than one project over a number of years. This is modeled as part of the board’s cost-of-ownership in the CCS. Regardless of cost, the extra time and effort involved in managing a BoM purchase compared to the purchase of a single off-the-shelf board is worth considering.

There are also economies of scale from which mass producers of commercial FPGA boards are able to benefit, resulting in lower prices for each final user. For example, the cost per device of the FPGAs will be lower to an external mass producer

compared to buying them yourself in low-volume from your local distributor. In the CCS we can enter all material and component cost as a single overall sum which will then be used for other calculations.

Finally, for the bare-base PCB, assuming your manufacture is out-sourced, there will be a non-recurring engineering (NRE) or tooling cost for the board. In addition, the cost for the bare PCB is also significant enough for both to warrant separate entries in the CCS. As an illustration, a typical bare board cost for a four-FPGA board is approximately \$2,000 in low volume with a further \$2,000 to \$3,000 NRE which should also be amortized.

Component and NRE example

We enter the approximate cost for the components including the PCB manufacture, the board itself, the FPGAs, and other components. The full BoM will be well understood for your own boards but for this example, we have estimated full component and board costs as shown in Figure 178.

Figure 178: Estimates for material costs entered into CCS

Component and PCB costs	
NRE for PCB	\$2,500
Manufacture cost for bare PCB	\$1,500
Cost of components (per board)	\$32,000
Yield after test	70%
Total build to yield required boards	5

Total BoM price will vary depending upon the number of devices bought, minimum order quantities and discounts negotiated with vendors of the more important components on the board, for example the FPGAs and connectors. Of the \$32,000 shown in our example, nearly half of that would be the cost of the four FPGA components themselves.

These components will need to be assembled onto the board and whether or not this assembly is performed in-house or contracted out, there is an associated cost to its setup. This may be an NRE at the external assembly house, or a less tangible in-house personnel/equipment cost. The CCS makes different calculations depending on your choice. In our example, we calculate an assembly cost per board of between \$1,100 and \$1,700 depending upon time during which in-house machinery is employed or external outsourcing is used.

These costs do not include board functional testing, which are accounted for elsewhere.

B6. Direct cost: yield and wastage

FPGA prototype boards are very complex and it is probable that the PCB design will require some rework, so the CCS takes account of possible re-spin costs and delays. Commercial board providers are able to amortize such re-spin costs over their entire production volume, whereas, in-house re-spin cost is borne by the awaiting SoC project.

In addition to board re-spin, it is also highly unlikely that there will be 100% yield of the boards after manufacture and assembly. Prototype boards are very sophisticated and have a large number of components and complex surface mount packages. Boards which do not initially work must have their failure diagnosed and remedial action must be taken. Automated production test methodology is often not economical for the very small volume builds and so the test and rework is usually performed by hand, often by the same experts that originally created the design. It is common to overbuild boards so that adequate boards pass initial test with minimal delay.

It is an unnecessary risk to build exactly as many boards as required and delay the prototyping project because engineers are occupied reworking the in-house boards rather than using the finished boards to verify the SoC. To put some figures on this, a typical yield after board population and initial test is 70%; so this is mitigated by building ten boards in order to quickly yield seven working boards. Commercial board vendors may have such rates for early production runs but they make incremental design and manufacture tweaks in order to remove recurrent faults, eventually raising the yield closer to 100%.

Additionally, the vendor has time for remedial rework on non-yielding boards as projects are not delayed and all boards will eventually be able to reach working stock. Finally, commercial board vendors have the production volume which allows more automation and faster yield to stock time. All yield improvement reduces wastage and cost.

We may find that delivery from working stock vs. bespoke manufacture and test is the most important differentiator in our make vs. buy discussion.

B7. Direct cost: support and documentation

One obvious way to reduce the above costs per board is to build enough boards so that they may be used for more than one project, thus reducing the cost per board per project. This may require a more flexible design and more robust fabrication in

order to allow storage and re-deployment. Before the rise of today's commercial board vendors, major users of FPGA prototyping had already built up their own internal board standards for reuse across multiple projects. Along with common hardware and interfaces made possible by this approach, the prototyping engineering expertise is also centralized and made available as a design service within the organization.

An extra task in such centralized organizations is the support and documentation of the boards for remote deployment. For example, the board may have been designed for use in an SoC project in the USA but reused for a quite different project by a team in Asia. Documentation must be adequate to allow such reuse and the original board designers should be available to support regular reuse as required. For this reason, it is normally only very large multi-national design teams that can afford such an investment in company-wide standards. Once established, prototyping teams may offer their valuable experience widely within their companies, however, it does not matter if the boards are developed in-house or sourced externally for this prototyping expertise to be valuable.

We have now considered the direct costs in producing or sourcing FPGA prototyping boards. Let us go on to discuss the previously mentioned business costs in turn. Some readers might say that these business costs are intangible and less under the control of Engineering department, however, they are often the deciding factor in the success or otherwise of a prototyping project.

B8. Business cost: time

How long does it take to design and make an FPGA prototype board? A typical 25 layer board with four large FPGAs in BGA packages, having maximum interconnect and necessary ancillary circuits (e.g., configuration and supervision functions) is a significant engineering challenge in its own right. Approaching this challenge with inadequate resources or insufficient experience often results in delay or failure; this is not a part-time job!

However, assuming we have the experience and sufficient access to the resources (e.g., PCB layout department) then we may estimate the time required. The estimate is broken into discrete tasks as follows:

- Task 1: Circuit design
- Task 2: PCB layout
- Task 3: PCB manufacture
- Task 4: Assembly set-up time including BoM purchase
- Task 5: Assembly time
- Task 6: Test time
- Task 7: Rework time

Some of these tasks may be done in parallel, for example tasks 5, 6 and 7 would be like a “production” line with different boards at each stage at any one time. Other tasks are clearly sequential by necessity. It is also possible that task 4 might be started soon after task 1 is complete, assuming, of course, that the same people are not performing both tasks.

Allowing for parallel work, we arrive at the best-case project dependency shown in Figure 179.

Figure 179: Typical critical path for in-house board project



Once again, readers are encouraged to use their own in-house data, but for illustration, the authors are aware of typical design times of four weeks, PCB layout of eight weeks, manufacture two weeks, and test of one week. It is necessary to justify the PCB layout figure of eight weeks so let us consider this in particular.

High-performance FPGA prototyping boards are not simple. We are aiming to run very many traces between the FPGAs and other resources (connectors, memories etc.). The FPGAs are shipped in BGA packages with approximately 1,700 pins each, which lead to many thousands of traces being required on the board. It soon becomes apparent that a many-layered board is necessary which is pushing the limits of some automated PCB place & route tools. We also consider that, to maximize performance, clocks and resets must be cleanly distributed to all parts of the boards with minimal corruption or crosstalk. There is also the problem of delay

matching on the board traces in order to balance the effect of signal distribution between FPGAs. Considering these criteria and other constraints, a layout in eight weeks is very reasonable.

Using the above approximations, we arrive at a best-case estimate of 15 weeks for in-house FPGA board development. How does this compare with a typical duration of an SoC prototyping project?

Analyzing survey data provided during users' group meetings and dedicated on-line surveys, the authors have collated some typical information regarding the duration of FPGA projects. For example, Table 38 shows data summarized from a FPGA users' survey performed in October 2007.

Table 38: Comparing prototyping and non-prototyping FPGA project durations

Purpose or project	Duration of project (weeks)					Total projects
	up to 13	13-26	26-39	39-52	over 52	
ASIC or SoC prototype	25	46	20	9	3	103
Other	10	9	1	3	1	26
Production	30	40	32	19	16	137
Total	65	95	53	31	20	264

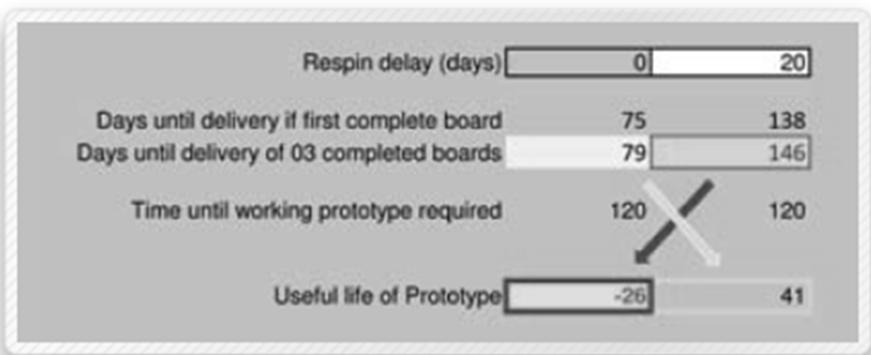
Within the granularity limits of this data, we can calculate that the average duration of a prototyping project is between 15 to 27 weeks. It is clear that the 15-weeks board development time estimated above does not fit well with our survey results. If the board were taken as the first task in a prototyping project, then the length of the project could double. If there are enough engineers available then board development and some of the design preparation can be performed in parallel. However, there is added risk if the board design is fixed before the SoC RTL is first mapped to FPGAs. We will consider this and other risks in the next section.

Example: time estimates

All of the above figures are used to calculate the cost but they also provide a coarse project timeline. Of course, the CCS must only be used as a rough rule-of-thumb before entering into proper timeline analysis as part of correct project management. In Figure 180 we see the CCS time estimates based on all the above data entered for our example. The spread of best-case and worst-case results is very wide.

Unfortunately, the time allowed for completion of the prototype project is a

Figure 180: Comparison of estimated useful lifetime of boards (from CCS)



constant. In our example, we can already see that if the project is started with in-house board development, then the useful time for actually using the prototype to verify the SoC can quickly disappear. In this example, there is a risk that the boards will not be available until well after the planned end of the project.

B9. Business cost: risk

The aim of our FPGA-based prototyping methodology is to reduce risk in an SoC project; therefore we should not be adding further risk or delays by cutting corners on verification.

Delays in the prototype causes delays in the SoC verification overall and/or delay in the SoC sign off and follow-on delays in the product's final introduction. It is widely accepted that release postponement will reduce a product's total revenue in the end-market so risk is counted as a related business cost in this section i.e., raised risk means that market-reducing delays are more likely to happen.

For this analysis we will compare the risk in building in-house boards vs. buying off-the-shelf. For example, delays may occur because of limited supply of off-the-shelf boards or because of a longer than expected development or manufacture time of in-house boards. On the other hand, external board suppliers must also be able to deliver to their quoted schedules so this is not a one-sided consideration. Let us consider how risk might be reduced.

Excess capacity reduces risk: let us continue the previous discussion about saving time by designing the board and preparing the SoC design in parallel. It is possible that the FPGA capacity or interconnect requirements might be incorrectly estimated too early in the project; for example prior to partitioning. Of course the SoC design

itself may also change as the SoC project matures but this would be a risk in either the in-house board or off-the-shelf board. More realistic estimates would yield even higher per-project costs for the in-house boards.

The risk in starting the in-house FPGA board design early may be mitigated by making the board design as flexible as possible. This contradicts the natural temptation to design the board with topology to match the top-level blocks of the SoC, e.g., “this big block in that big FPGA; this smaller block in that smaller FPGA.”

Instead, we should try to provide some margin in the available FPGA resources on board. A good rule of thumb is to place twice as much FPGA resource on the board as the initial synthesis and place & route runs suggest. Skimping on the FPGAs used is a false economy compared the increased risk placed into the SoC project by doing so.

Interconnect flexibility reduces risk: the interconnect traces between the FPGAs should be ample and general rather than guided by the top-level signal buses in the SoC design. Top-level design interconnect can change if new top-level functions are added to the design so the board-level interconnect must be able to follow suit. If it cannot, then unnecessary project delay may occur while a new partition of the design is found, perhaps requiring that signals be multiplexed onto the same interconnect.

Whether or not the interconnect resource needs to be performed remotely using programmable links, or locally and manually altered by using physical connectors, depends on the requirement for the boards. If it is to be used almost simultaneously for multiple projects, perhaps like some kind of “prototyping farm,” then it would need to be remotely programmable, probably using some kind of in-line signal switching between FPGAs.

Alternatively, it may be that the platform will be configured once per project with the right FPGA resources with the interconnect established and the necessary peripherals in place. In this case, the interconnect “switching” can be manual which simplifies the board design and may allow higher performance.

Peripheral de-coupling reduces risk (and increases reuse): as mentioned previously, in-house boards may be designed for a single project or might be expected to be used across multiple projects. In the former case, it is tempting to place all the necessary hardware peripherals needed to match the functions instantiated inside the SoC, for example, the RAM, PCI, video, and co-processor cores in chip form, mounted and connected into the board in the same way that they are in the SoC. The risk introduced is the same as for the capacity and interconnect cases mentioned above. If the SoC topography or peripheral content is changed (for example, a second PCI port is added) then it may take many weeks to rework the board to add the new function. A modular approach is clearly advisable, so that

daughter cards may be mounted on the main FPGA boards in order to provide the necessary extra peripheral functions.

Those expert teams providing many in-house boards across multiple projects universally adopt a modular arrangement of motherboards and separate peripherals which may be designed quickly and with less error.

If the peripheral boards are connected together with a motherboard using a standard connector arrangement then this greatly increases the options for their reuse in other projects. Some commercially available FPGA prototyping systems follow this modular approach as well and have built up a large selection of optional peripheral functions available as daughter boards. This allows us to select and assemble the required resources for the project and then reassemble and augment for the next project. This also reduces the risk that late changes in the SoC design cannot be quickly accommodated by the prototype board.

Expert advice reduces risk: if this is your first prototyping project, then it is advisable to employ an external consultant or allocate your best staff FPGA experts to help with SoC design preparation. They will be able to give early appraisal of the necessary resources, interconnect, and flexibility required and ensure that late project changes do not render the entire prototype unusable. They should not only be able to provide high-quality guidance to the FPGA board designers, but also appraise whether commercially available boards will meet the project requirements.

B10. Business cost: opportunity

Having considered time and risk, we should also include the opportunity cost for developing boards in-house. Opportunity cost can be defined as that benefit which has not been realized elsewhere because resources (i.e., engineers in this case) are allocated to a given project. In the CCS, opportunity cost is expressed as a missed ROI. You can enter the normally expected revenue (or other output) generated by the engineers as a percentage of their cost. For example, each engineer may otherwise generate \$2.00 of final revenue for each \$1.00 of cost (i.e., salary etc.) resulting in an ROI of 200%. This is realistic because design of FPGA prototype boards takes a great deal of FPGA design expertise which would also be very valuable elsewhere.

B11. CCS worked example results

Given the variables and constants we have used in our example throughout this section, the main result from the CCS will be the cost comparison. The result panel for the CCS is shown in Figure 181.

The cost is either per board, such as test, or per project, such as the PCB layout

Figure 181: Design example cost results given by CCS

In-House Cost Analysis		
Direct Costs	Min	Max
Total Resource personnel cost	\$30,000	\$46,596
Total component and pcb cost	\$170,000	\$170,000
Useful Boards	3	3
Resource+component cost per board	\$66,667	\$72,199
Assembly cost per board	\$3,000	\$4,333
Direct Cost per board	\$69,667	\$76,532
Related Costs	Min	Max
Total opportunity cost for personnel	\$37,500	\$58,245
Total cost of ownership	\$0	\$0
Total Related Costs	\$37,500	\$58,245
Useful boards	3	3
Related Costs per board	\$12,500	\$19,415
Total cost for in-house board	Min	Max
Lifetime cost per board	\$82,167	\$95,947

which then has to be amortized across the total number of boards in order to arrive at a per board amount; obviously, the larger the number of boards, the smaller the cost of each board. Commercial board vendors can take advantage of this “economy of scale” and to pass on these economies to customers. We should expect that for low volumes, it is cheaper to buy boards than to make them in-house.

The example prototyping project calls for low volume (only three boards) and with the previously mentioned costs, we see that the best-case cost for each board would be around \$82,000 or as much as \$96,000 if it took a little longer for each task and a PCB re-spin was required. We have purposely used low estimates and values in this example; more realistic estimates would yield even higher per-project costs for the in-house boards.

If you work with a spreadsheet like the CCS or a much more sophisticated project management tool, you can alter any of the variables to see what effect this will have on the real board costs. However, it is important to keep the board cost in perspective to the other costs and risks and how that fits into the overall SoC project timescales.

B12. Summary

For any given project there is a threshold of board volume beyond which it will be more economical to make boards in-house. Beyond the economic arguments there may be technical merit in making boards in house, for example to meet a certain form factor or topology requirement. Using a spreadsheet like the CCS, we can be sure to expose all the cost and timescale implications but the technical requirements, as explored during this chapter, may be a deciding factor. Most important of all, in the end, is time. It is critical to FPGA-based prototyping that the SoC project does not depend upon early delivery of new in-house boards.

Generally, the minimum-risk approach does not involve making boards in-house, however, the alternative of finding the right commercially available boards to meet the same need in doing so may be very high.

References and Bibliography

The authors recommend the following reference sources. Readers of the pdf version of this book will be able to link directly from each reference. Some of the links may require access to the Synopsys® SolvNet® Technical Support website.

FPGA and prototyping design information

“Advanced FPGA Design: Architecture, Implementation and Optimization”

Kilts, Stev. 2007

John Wiley & Sons. (hardcover) ISBN 978-0-470-05437-6

“CHIPit SCE-MI 2.0 Implementation Handbook, V1.9”

Synopsys. November 2010

solvnet.synopsys.com/dow_retrieve/E-2010.12/chipit/handbooks/chipit_scemi.pdf

“HDL Bridge and Signal Tracker, V3.3”

Synopsys. 2010

solvnet.synopsys.com/dow_retrieve/E-2010.12/chipit/handbooks/hdl_bridge.pdf

“PCIe and SATA AHB core with RIO datasheet”

Lewis, Torrey with Shehyrar Shaheen, Peter Gillen, Synopsys, October 2009
(available on request from Synopsys)

“Rapid System Prototyping with FPGAs: Accelerating the design process”

Cofer, RC with Benjamin F. Harding, 2006

Newnes. (softcover) ISBN 978-0-7506-7866-7

“Synchronizing Reset”

Synopsys Application Note number SH300100-01

(available on request from Synopsys)

“System ACE CompactFlash Solution”

Xilinx. October 2008

http://www.xilinx.com/support/documentation/data_sheets/ds080.pdf

Standards and other background information

“Low Power Methodology Manual”

Keating, Michael with David Flynn, Robert Aitken, Alan Gibbons and Kaijian Shi
Springer. (Hardcover) ISBN: 978-0-387-71818-7

“Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual, V2.0”

EDA Industry Working Groups – Accellera. January 2007

<http://www.eda-stds.org/itc/scemi200.pdf>

“SystemC: From the Ground Up”

Black, David C. with Jack Donovan.

Springer. (softcover) ISBN 978-0-387-29240-5

“Verification Methodology Manual for SystemVerilog”

Bergeron, Janick with Eduard Cerny, Alan Hunter and Andrew Nightingale, 2006

Springer. (Hardcover) ISBN: 978-0-387-25538-5

“VMM Hardware Abstraction Layer (HAL) User Guide, V 2.1”

Synopsys. December 2009

http://vmmcentral.org/pdfs/vmm_hw_abstraction_guide.pdf

“VMM for System Verilog online directory”

VMMCentral. 2010

<http://www.vmmcentral.org/resources.html>

“Unlocking Android, second edition”

Ableson, W. Frank with Robi Sen

Manning. ISBN 978-1-935-18272-6

Xilinx FPGA device information

“Virtex-6 Family Overview”

Xilinx. January 2010

http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

“Virtex-6 FPGA Clocking Resources User Guide UG362, V1.5”

Xilinx. August 2010

www.xilinx.com/support/documentation/user_guides/ug362.pdf

“Virtex-6 FPGA Memory Resources Users Guide UG363 VI.5”

Xilinx. August 2010

www.xilinx.com/support/documentation/user_guides/ug363.pdf

“Virtex-6 User’s Guides, online directory”

Xilinx. undated

<http://www.xilinx.com/support/documentation/virtex-6.htm#131591Xilinx>

Synopsys prototyping platforms information

“CHIPit Platinum Edition Handbook, V4.6”

Synopsys. 2009

solvnet.synopsys.com/dow_retrieve/E-2010.12/chipit/handbooks/platinum.pdf

“*CHIPit Iridium Edition Handbook, V2.6*”

Synopsys. 2009

solvnet.synopsys.com/dow_retrieve/E-2010.12/chipit/handbooks/iridium.pdf

“*HAPS-60 Series of High-performance FPGA-Based Prototyping Systems*”

Synopsys. 2010

www.synopsys.com/Systems/FPGABasedPrototyping/Pages/HAPS-60-series.aspx

“*UMRBus Communication System Handbook, V3.5*”

Synopsys. December 2010

solvnet.synopsys.com/dow_retrieve/E-2010.12/chipit/handbooks/umrbus.pdf

FPGA and prototyping tools information

“*Certify Partition Driven Synthesis User Guide*”

Synopsys. December 2010

solvnet.synopsys.com/dow_retrieve/E-2010.12/certify/user_guide.pdf

“*CHIPit Manager Pro Handbook, V3.5*”

Synopsys. June 2010

solvnet.synopsys.com/dow_retrieve/E-2010.12/chipit/handbooks/chipitpro.pdf

“*Identify User Guide*”

Synopsys. September 2010

solvnet.synopsys.com/dow_retrieve/E-2010.12/identify/user_guide.pdf

“*Synopsys FPGA Synthesis User guide*”

Synopsys. December 2010

solvnet.synopsys.com/dow_retrieve/E-2010.12/synplify/user_guide.pdf

“*Xilinx Implementation Flow Overview*”

Xilinx. 2010

http://www.xilinx.com/itp/xilinx8/help/iseguide/html/ise_fpga_design_flow_overview.htm

“*Xilinx ISE Manuals online directory*”

Xilinx. 2010

<http://toolbox.xilinx.com/docsan/xilinx7/books/manuals.pdf>

Other tools information

“*DesignWare Developers Guide*”

Synopsys. December 2010

https://www.synopsys.com/dw/doc.php/doc/dwf/manuals/dw_developers_guide.pdf

“*DesignWare Reference Online*”
Synopsys. 2010
<http://www.synopsys.com/dw/dwlibdocs.php>

“*DesignWare System-Level Library online documentation*”
Synopsys. 2010
www.synopsys.com/Systems/SLModels/Pages/default.aspx

“*Practical Perforce*”
Wingerd, Laura. November 2005.
O'Reilly Media. (softcover) ISBN 978-0-596-10185-5

Board design information

“*Electromagnetism: Theory and Applications*, 2nd Edition”
Lorrain, Paul and Dale R. Corson. April 1990
W.H. Freeman.

“*High-speed Signal Propagation Advanced Black Magic*”
Johnson, Howard and Martin Graham, March 2003
Prentice Hall. ISBN 0-13-084408-X

“*RF Circuit Design, Theory and Applications*”
Ludwig, Reinhold and Pavel Bretchko, January 2000
Prentice Hall. ISBN 978-0-311-47137-5

“*Right the First Time: A Practical Handbook on High Speed PCB and System Design*”
Ritchey, Lee W. with John Zasio and Kella J. Knack. June 2003
Speeding Edge. ISBN 978-0-974-19360-1

Acknowledgements

The authors would very much like to acknowledge the support and contribution of some excellent people, both inside our relevant organizations, at our partners and beyond (those named in the following list are either within Synopsys® or Xilinx® unless otherwise stated). The following people have acted in very many different roles from partner and advisor, through reviewer up to and including contributor of written material. Some are colleagues, some managers and many are technical consultants and R&D professionals inside our two organizations but most noteworthy of all are the industry leaders who are performing FPGA-based prototyping in labs around the world every day, to this last group is this book dedicated.

The FPMM project started with extensive interviews with R&D professionals in our own labs. We have a great deal of internal expertise on FPGA-based prototyping and following folks got us off to a flying start; **Alf Larsson, Anders Nagle, Andreas Jahn, Bo Nilsson, David Svensson, David Taylor, Jonas Magnuson, Jonas Nilsson, Martin Johansson, Mathias Svensson, Matthias Lange, Michael Roeder, Michael Schmidt, Ronny Franz, Steffi Sier and Sven Lasch.**

Then over the duration of the project many other R&D advisors have joined to make their contribution including, **Antonio Costa, Kevin Smart, Nithin Kumar Guggilla, Peter Gillen, Pradeep Gothi, Rajkumar Methuku, Ramanan Sanjeevi Krishnan, Sanjana Nair, Sivaramalingam Palaniappan, Torrey Lewis and Vidyullatha Murthy.** To these hard-working people and those above, we offer our thanks for finding the time to help us with the examples, diagrams and insight.

Outside of R&D we have just as a strong a technical expertise in the applications consultants teams and services teams within both Xilinx and Synopsys. Some of these also became long term advisors in our bi-weekly calls discussing many aspects of the use of FPGA-based prototyping in our locales, some provided examples or reviewed drafts. In particular, we want to acknowledge **Andy Jolley, Bob Efram, Chuck Banken, David Castle, Davin Lim, Derek Johnson, Tom Fairbairn, Doug Fisher, Edmund Fung, Frederic Rivoallon, Jim Heaton, Jon Talbot, Mark Nadon, Mike Cole, Peter Calabrese, Pradeep Kumar, Robert Perry, Tom Fairbairn, Tom West, Will Cummings and Chelman Wong.** To you all, we offer our sincere thanks for your encouragement.

About halfway through, we invited industry leaders to join the FPMM Review Council in order to provide feedback on the drafts which were becoming available at that time. Some went further and took part in interviews and even donated content for the book that you see before you. To all the following reviewers and contributors, we offer our thanks for sharing their insight as leaders in the field of FPGA-based prototyping: **Andrew Gardner (ST-Ericsson), Andy O'Brien (Intel Corp.), Brian Nowak (LSI Corp.), Clay Douglass (Intel Corp.), David Stoller (Texas Instruments), Fernando Martinez (NVIDIA Corp.), Gavin Dolling**

(Displaylink Ltd.), German Fabila Garcia (Intel Corp.), GN Choudhary (STMicroelectronics), Graham Deacon (Imagination Technologies Ltd.), Helena Krupnova (STMicroelectronics), Javier Jimenez (Marvell Hispania), Joel Sandgathe (Microvision Inc.), Justin Mitchell (BBC Research & Development), Sabyasachi Dey (Qualcomm), Scott Constable (Freescale Semiconductor), Spencer Saunders (ARM Ltd.), Steve Ravet (ARM Ltd), Tim Stretch (ST-Ericsson) and Yuji Yoshitani (Fujitsu).

A book is a long project and involves many tasks to bring to completion including editing, formatting, proofreading, drawing, artwork, project planning, funding and copy writing, administration and more besides. We would like to acknowledge the following colleagues both within Synopsys and Xilinx and amongst our partners, who are a credit to their profession: **Chris Dace** (copy, proofing), **David Abada** (copy), **Frank Schirrmeister** (copy), **Irene Economou** (planning), **Jeff Baran** (website), **Lisa Rivera** (production), **Nancy Weiss** (planning), **Phil Dworsky** (publishing, legal), **Rena Ayeras** (artwork, formatting), **Rick England** (copy), **Sheryl Gulizia** (PR), **Taeko Tanaka** (artwork), **Tom Sidle** (copy), **Tony Smith** (Perforce info); and **Carol Amos** (tea, smiles).

There are also some other colleagues in our teams upon whose expertise we were privileged to be able to draw. Thanks to all of you for helping to bring the FPMM to completion: **Angela Sutton, Dave Vornholt, Greg Lara, Gunnar Scholl, Janick Bergeron, Jeff Garrison, John Simmons, Karen Bartleson, Larry Vivolo, Laura MacIvor, Markus Willems, Mick Posner, Mike Lux, Mike Santarini, Phil Morris and Robert Smith.**

And finally, last but definitely not least, we want to salute the management of both Synopsys and Xilinx who had the vision, the faith and above all, the budget, to see this project through to completion. To roll of honor includes **Andrew Dauman, Bruce Jewett, David Tokic, Ed Bard, Gary Meyers, George Zafiroopoulos, Heiko Mauersberger, John Koeter, Patrick Dorsey, Rajiv Maheshwary and Steve Trimberger.**

Very many thanks to all of the above.

What's next?!

Austin, Doug and René

Figure Sources

The following figures are used by permission:

- Figure 5: Android™ Developers website
- Figure 6, 7 & 8: International Business Strategies, Inc (Los Gatos, CA)
- Figure 9 &10: Semico Research Corp. © 2010
- Figure 11 & 12: International Technology Roadmap for Semiconductors
- Figure 15: Frank Abelson, from his book “Unlocking Android” (see references)
- Figure 16: Freescale® Cellular Products Group, (Austin, TX)
- Figure 18: DS2, (Valencia, Spain)
- Figure 19: BBC Research and Development, (London, UK)
- Figure 20 – 25: Xilinx®, Inc. (San Jose, CA)
- Figure 34: Xilinx®, Inc. (San Jose, CA)
- Figure 37 - 39: Xilinx®, Inc. (San Jose, CA)
- Figure 56 & 58: Xilinx®, Inc. (San Jose, CA)
- Figure 85: Xilinx®, Inc. (San Jose, CA)
- Figure 123 & 124: ARM Ltd., (Cambridge, UK)
- Figure 167 & 168: Applied Micro Circuits Corporation (AMCC) from Linley Technologies Seminar entitled “Programmable Devices for Network Systems” held in San Jose, CA, Nov 1st, 2006

All other figures and tables: Synopsys inc. or public domain sources

Glossary of key terms

ASIC	Application Specific Integrated Circuit. A chip which is custom designed for a specific use and purpose.
ASSP	Application Specific Standard Product. A chip designed for a specific application but meant for use in multiple applications.
BGA	Ball Grid Array. A surface-mount packing technique.
BIST	Built-in-Self-Test. Usually for testing memories and other regular structures
BitGen	The command-line used for configuration step performed by Xilinx ISE
Bitstream	A bitstream is used to program an FPGA device. It contains information to configure the FPGA routing and logic resources as designed by designer.
Block RAM	A block of random-access memory built into the device, as distinguished from distributed, LUT-based random-access memory.
BUFGCE	A Xilinx® primitive which is basically used for clock management. Global clock buffer is gated with a clock-enable signal. If clock enable is disabled the clock will also be disabled.
BUFGMUX	Used to switch between clocks without glitches.
Carry logic	Mainly used to implement arithmetic logic functions and exists in each slide and runs along each column of CLBs as well as the top and bottom CLBs.
Certify®	Name of the Synopsys FPGA-based prototyping and partitioning software tools.
ChipScope™	ChipScope is the integrated logic analyzer add-on to the Xilinx® ISE® software.
Core	A colloquial name for a complex block of IP.
DCM	Digital Clock Manager. A design element which provides multiple functions. It can implement a clock-delay locked loop, a digital frequency synthesizer, digital phase shifter, and a digital spread spectrum.
DDR	Double Data Rate uses both edges of a clock to capture data.
EDIF	Electronic Design Interchange Format. An industry-standard netlist format.

Footprint	The shape, pin names, and functionality of a library macro or component.
Foundry	A silicon wafer fabrication facility which offers production services to other parties.
Gate Array	An early and still common form of ASIC chip. This requires the use of bespoke masks for the last few stages of chip fabrication to be tailored to a specific design.
Global Buffers	Low-skew, high-speed buffers that drive large fanout signals, usually clocks, across an FPGA.
HDL	Hardware Description Language. A language used for modeling designing and simulating hardware. The two most common HDL languages are VHDL and Verilog.
IO blocks	The input/output logic of the device containing pin drivers, registers and latches, and 3-state control.
IO pads	Input/output pads that interface the silicon device with the pins of the device package.
IBUF	A circuit which acts as a protection for the chip, shielding it from eventual current overflows.
ICE	In-circuit Emulation.
ISE®	Integrated Software Environment, the overall name for Xilinx® FPGA tools, including project management, place & route, bitstream generation and debug. Pronounced as 'ice'
ISERDES	Input Serializer/Deserializer. A dedicated source synchronous IO architecture.
JTAG	Joint Test Action group is an IEEE 1149.1 standard test access port and boundary scan architecture.
LOC	Location constraint. Used to lock pin locations or to place logic in specific locations in the FPGA.
LVDS	Low Voltage Differential Signaling. Differential signaling is a method of transmitting information electrically by means of two complementary signals sent on two separate wires. This offers better immunity to ambient noise and faster signal propagation.
Macro (1)	A slightly out-of-date term for a component made of nets and primitives, FFs, or latches that implement mid-level functions, such as add, increment and divide. Soft macros and relationally placed macros (RPMs) are types of macros.

Macro (2)	A universal control embedded into the RTL and visible from any point during compilation. Often defined in Verilog using `define command.
MMCM	Mixed-Mode Clock Manager. Multi-output configurable block which includes PLL and phase shifters to give fine-grain control of clocks within a Xilinx® FPGA.
NCD	A Native Generic Database file describes the logical design reduced to Xilinx® primitives.
Netlist	A text description of the circuit connectivity. It is basically a list of connectors, a list of instances, and, for each instance, a list of the signals connected to the instance terminals. In addition, the netlist contains attribute information.
Pad-to-pad delay	A combinatorial path which starts at an input of the chip and ends at an output of the chip. The pad-to-pad path time is the maximum time required for the data to enter the chip, travel through logic and routing, and leave the chip. It is not controlled or affected by any clock signal.
PCIe	Peripheral Component Interconnect - Express: An international standard for fast serial interconnection.
Pin locking	Process by which a signal is given a location on a specific FPGA package pin.
Place & route	Two major processes involved in back-end tools deciding which resources on an FPGA will be used to implement each part of the netlist. and then routing the signals between them.
PLL	Phase locked Loop is an analog clock-locking circuit. Compares two clock signals and aligns the two. Used for synchronizing and zero-delay buffering plus division and multiplication of frequency.
Primitives	The most fundamental design elements in the Xilinx® libraries, sometimes referred to as BELs, or Basic Elements. Primitives are the design element "atoms" and can be combined to create macros. Examples of Xilinx® primitives are the simple buffer, BUF, and the D FF with clock enable and clear, FDCE.
RLOC	Relative Location Constraints . Used to group logic elements together to reduce routing delays in the design. Logic elements with RLOC can be moved but must maintain the same relative placement. This prevents

routing delays appearing between the elements. RLOCs are used to create (relatively placed macros (RPMs).

RTL	The short name given to hardware description code written at the Register Transfer Level, rather than at a system level or gate level. Often used colloquially to denote the source for a design.
Skew	Delay introduced differentially into one signal with respect to another.
TRACE	The Timing Reporter And Circuit Evaluator. A command-line utility for performing static timing analysis of a design based on input timing constraints.
UCF	User Constraints File. A format for feeding constraints and other controls into the Xilinx® ISE® tools.

Index

- access system
 bus-based, 339
- accuracy, 13
- ADC, 56
- adders/subtractors, 62
- adopting standard boards, 140
- alarms limits, 56
- Amdahl's law, 383
- AMS design, 402
- analog
 auxiliary circuit, 132
 blocks outside FPGA, 95
 design elements, 96
 model on-board, 94
- analog IP, 289
- Analog-to-Digital Converter. *See* ADC
- analyzers, 345
- Android
 SDK, 8
 software, 28
- application software, 11
- application specificity, 391
- architectural exploration, 39
- architecture exploration, 10
- arithmetic functions, 43
- assign clocks, 239
- assign signal traces, 234
- assign signals to traces, 235
- assigning blocks
 impact analysis, 230
- asynchronous loops, 286
- asynchronous multiplexing, 261
- automatic partitioner
 multiplexing, 239
- automatic partitioning, 239
- automatic trace assignment, 236
- automotive, 397
- auxiliary circuits, 132
- back-end, 107
- back-end flow, 75
- back-end tools, 73, 77
- BBC, 36
- BFM, 369
- bit-write enable mask, 211
- black box, 309
 FPGA synthesis, 188
 inferred soft IP, 301
 IP instantiation, 293
 loops, 280
 partition space, 239
 prototyping, 168
 RTL design, 53
 SoC, 192
 soft IP instantiation, 301
 timing information, 240
 verification, 283
- block
 IP, 19
 wrapper, 282
- block scaling, 96
- BlockRAM, 47
 read only memory, 212
- BlockRAM configurability, 47
- BlockRAMs, 47
- board
 physical stiffness, 358
 board connectors, 359
 board enclosure, 359
 Board engineering cost, 424
 board mounting, 358
 holes, 358
- board routing, 138
- board support package. *See* BSP
- board-level environment, 329
- boards
 mother - daughter, 117
 power distribution, 130
- branching macro, 206
- break even
 return of investment (ROI), 6
- bring-up cost, 13

<p>bring-up design guidelines, 314</p> <p>BSP, 29</p> <p>BUFG global buffers, 245</p> <p>built-in IP, 54</p> <p>built-in security, 344</p> <p>bus functional model. <i>See</i> BFM</p> <p>bus-based design access, 336</p> <p>bus-based instrumentation, 336</p> <p>cable umbilical, 116</p> <p>cables download, 137 external, 360 internal, 360 system access, 360</p> <p>capacitors power surges, 133</p> <p>capacity, 13</p> <p>CAPIM, 338</p> <p>CAPIMs UMRBus, 338</p> <p>CCS, 423</p> <p>Certify, 224 assign block destinations, 233 partitioning - port names, 198 partitioning tool, 72 resource estimation, 225 trace placement, 235</p> <p>Certify Pin Multiplier) assist. <i>See</i> CPM</p> <p>check external chipsets, 343</p> <p>chip design building blocks, 4</p> <p>chip design requirements, 21</p> <p>chip development software, 18</p> <p>chip differentiation, 19</p> <p>chip support block, 168 clock, 174 elements, 167 top level elements, 168</p> <p>CHIPit debug tool, 81</p>	<p>interconnect platform, 152</p> <p>prototyping system, 80</p> <p>ChipScope, 73 debug, 82 debug tool, 84 implementation flow, 82</p> <p>circuit auxiliary, 132</p> <p>CLB, 44</p> <p>clock distribution delays, 124 gating, 65 generation logic, 285 generation modules, 250 global lines, 51 logic mapping, 61 MCMM, 98 network, 122 synchronization, 244</p> <p>clock delays matching, 124</p> <p>clock distribution, 105, 127</p> <p>clock generator, 246</p> <p>clock management tile. <i>See</i> CMT</p> <p>clock networks, 285 partitioned, 245</p> <p>clock rate, 344</p> <p>clock rate factors, 103</p> <p>clock resources on-board, 123 SoC design, 99</p> <p>clock RTL gating, 170</p> <p>clock scaling, 127</p> <p>clock skew and uncertainty, 244</p> <p>clock synchronization, 244</p> <p>clock tree, 246</p> <p>clocking elements, 127 recommendations, 126</p> <p>clocking and timing, 95</p> <p>clocking guidelines, 284</p> <p>clocking resources, 98</p> <p>clocks IO, 51 regional, 51</p>
---	---

clock-tree synthesis, 171
coarse partitioning., 229
code generators, 186
coding standards, 275
combinatorial boundaries, 242
combinatorial loops
 guidelines, 280
combinatorial paths, 279
commercial partitioning tool, 327
common control signal, 171
compile point. *See* CP
compile point flow, 347
complex memory, 207
configurable logic block. *See* CLB
configuration
 high speed, 339
 IO, 52
 IO pins, 51
 remote management, 341
 timing constraints, 45
configuration methods
 FPGA, 136
configuration modes, 137
configuration tool
 Hapsmap, 76
CONFPRO
 interface, 341
 transport data, 340
connectivity
 issues, 327
consistent impedance, 138
constrain resource usage, 239
constraint
 example, 76
 generation, 59
 logic placement, 64
 placement, 64
 system-level, 68
constraints
 floor planning, 77
 FPGA performance, 103
 legacy standard, 140
 pin location, 326
 relax timing, 109
 system partitioning, 383

TIMESPEC commands, 76
tool capabilities, 213
conversion
 gated clock, 172
cooling, 136, 360
core and peripheral IO, 306
core generation, 77
core voltage, 131
Coregen, 201
co-simulation, 366
cost
 development, 5
 FPGA resource, 100
cost comparison spreadsheet, 423
cost estimate
 time estimates, 432
cost example
 component and NRE, 428
counter test, 317
co-verification
 physical interfaces, 375
CP, 347
 boundary, 348
 incremental synthesis, 348
 nested, 347
 simultaneous synthesis, 349
CPM, 127, 253
CPU
 cores, 55
critical set-up, 344
cross module reference). More. *See* XMR
cross-connect matrix, 152
cross-module references. *See* XMR
cross-trigger, 81
current surges, 133
custom debuggers, 336
data plane functionality, 396
data streaming, 340
dataflow
 real-time, 27
daughter board mounting, 358
daughter board reference designs,
 317
daughter boards, 359

daughter card
 peripheral, 118
dc. *See* Design Compiler
DDR
 configuration, 52
 IOSERDES blocks, 262
debug
 checklist, 352
 embedded hardware, 334
 external interface, 327
 fault source, 333
 FPGA connectivity, 325
 hardware interface, 334
 hardware-software interface, 384
 higher bandwidth, 337
 instrumentation - ChipScope, 73
 interaction, 346
 IO pad configuration, 328
 lab prototype, 353
 multicore CPUs, 6
 plan, 346
 real-time signal analysis, 336
 RTL verification, 334
 software debugger, 342
 startup state, 332
 test points, 342
 tips, 343
 tool options, 89
 visibility, 134
debug insight, 14
debug techniques, 342
debug tool
 ChipScope, 84
debugger
 custom, 341
 embedded CPU, 341
 processor, 421
 software interface, 421
 software utility, 342
debugging tools, 77
decryption, 294
deferred interconnect, 155
delay
 IO, 52
deliverables, 270

deployment cost, 14
derived gated clock, 172
design
 board-level environment, 329
 clock tree, 246
 FPGA resources, 99
design access
 bus-based, 336
design adaptation for FPGA, 58
design capacity, 16
Design Compiler, 213
 naming rules, 214
 synthesis tool, 302
design constraints
 SoC design, 110
design example
 BBC, 36
 bit-enabled RAM, 209
 compare verification interfaces, 375
 external peripheral IP, 309
 FPGA prototype, 36
 HD data stream, 27
 HDL Bridge, 366
 monitor temperature, 135
 multiplexing and sampling FF, 256
 multiport RAM, 207
 packet processing sub-system, 411
 parameterizable RAM, 202
 partitioning tool - Certify, 72
 pipelined synchronous elements, 248
 prototyping real world data, 34
 RTL macros, 279
 SCE-MI 2.0, 372
 SoC constraint files, 213
 SoC memory, 194
 soft IP, 299
 soft IP block, 299
 synchronous multiplexing, 261
 TI - Texas Instruments, 411
 wireless headseat, 4
design flow, 287
design issues, 334

design iteration, 101, 105, 109, 241, 242, 327
design preservation, 350
design setup cost, 424
design start
 embedded processors, 18
design starts, 16, 401
DesignWare, 281
 function calls, 299
 soft IP, 300
development
 coordinate, 141
development cost, 5, 13
device properties, 213
device under test. *See DUT*
differential signals IO standards, 329
direct cost, 427
direct interconnect, 154
distributed RAM, 196
distribution
 development, 141
divide and conquer
 debug, 343
documentation, 274
 development, 141
double data rate. *See DDR*
drive current, 52, 329
DSP cascading, 62
DSP resource blocks, 97
DSP48E1, 48
DSP48E1 slice, 48
DUT, 12
DVB
 technical specification, 37
early customer demo, 356
ECC, 47
echo tests, 414
EDA
 partitioning, 222
electrical connectivity, 95
electronic system-level. *See ESL*
element
 replacement by inference, 189
elements
 process of including, 191

embedded CPUs, 401
embedded software, 17
embedded trace extraction, 336
emulator
 IICE, 85
 slow run speed, 9
emulators, 9
enclosure, 359
encrypted
 FPGA bitstream, 293
 FPGA images, 295
 keys, 293
 netlist, 293
 RTL, 311
encryption methodology, 294
engineering cost example, 426
engineering costs, 425
engineering resources, 111
error detection. *See ECC*
ESL, 39
estimating
 performance, 103
 required resources, 106
 resources, 92
ethernet channels, 54
execution control, 14
execution speed, 13
exhibition demo, 356
external components, 327
external hardware options, 95
external interfaces
 prototyping, 103
fabrication, 42
Fast Binary Data Format. *See FBDF*
fast synthesis, 64
FBDF, 85
feasibility
 lab experiments, 35
field tests, 356
FIFO logic, 47
file list, 271
filter
 DSP, 98
 FIR, 317
 suitable candidates, 235

symmetrical, 48
final timing report, 217
FIR filter, 317
flexibility
 clock resources, 123
floor planning, 77
formal verification, 283
FPGA
 clock distribution, 50
 clock generation, 49
 clock resources, 49
 DSP resources, 47
 IO, 51
 logic blocks, 43
 logic elements, 62
 memory types, 46
 partitioning, 221
 prototype board, 424
FPGA board
 guidelines, 144
FPGA boundaries
 coherence, 332
 speed, 407
 visibility, 335
FPGA boundaries.
 multiplexing signals, 51
FPGA capacity, 160
FPGA clock resources, 172
FPGA clocking resources, 49–51
FPGA editing, 77
FPGA implementations
 common issues, 321
FPGA IO
 current drive, 133
FPGA IO pad configuration
 issues, 328
FPGA IO pins
 SoC design, 100
FPGA IP, 308
FPGA mapping
 external design blocks, 95
FPGA performance, 103
FPGA pins
 clock constraint, 240
FPGA platform, 361

flexibility, 121
hybrid prototype, 116
in-house standard, 142
interconnect, 119
modularity, 117
routing power, 128
set-up, 362
size and form, 115
topology, 115
FPGA platforms
 ready-made available, 112
FPGA project
 guidelines, 405
FPGA prototyping
 board capacity, 16
FPGA resources, 56
FPGA routing resources
 SoC design, 100
FPGA synthesis
 configure IO pad, 168
FPGA synthesis tool, 188
FPGA utilization, 108
FPGA utilization limits, 100
FPGA-to-FPGA connectivity, 104
FPMM website, 346
front-end, 107
FV. *See* formal verification
Gartner, 401
gated clock
 convertible conditions, 175
 derived, 172
gated clock conversion, 172
 automated, 174
 guidelines, 174
gated clock mapping, 65
gated clock timing, 324
gate-level netlist, 70
gating logic
 non-convertible, 176
generate memory, 201
getting started
 place and route scripts, 75
getting-started
 checklist, 91
gigabit transceiver blocks. *See* GTX

<p>gigabit transceivers, 53</p> <p>global clock, 246</p> <p>global clock lines, 51</p> <p>global clocks, 171</p> <p>global reset, 134, 139</p> <p>global start-up, 139</p> <p>golden reference, 330</p> <p>gray box, 193</p> <p>group blocks, 239</p> <p>group pins together, 239</p> <p>GTECH, 290</p> <p>GTX block, 53</p> <p>handling chip support elements, 168</p> <p>handling the IO pads, 168</p> <p>HAPS</p> <ul style="list-style-type: none"> 60 series boards, 159 connectivity options, 112 delay matchings, 124 Hapsmap configuration tool, 76 prototyping systems, 112 <p>HapsTrak II, 151</p> <p>hard IP</p> <ul style="list-style-type: none"> external, 308 formats, 298 resources, 95, 284 <p>hardware</p> <ul style="list-style-type: none"> AT models, 369 external options, 95 programmable, 17 <p>hardware stack, 4</p> <p>hardware-software</p> <ul style="list-style-type: none"> development, 9 integration, 28 verification, 40 <p>HDL</p> <ul style="list-style-type: none"> operators, 303 pragma, 302 Verilog XMR, 79 <p>HDL bridge, 366</p> <p>HDMI</p> <ul style="list-style-type: none"> dataflow prototype, 27 PHY, 27 Rx IC, 305 video data source, 308 <p>heat</p>	<p>dissipation, 136</p> <p>sink, 136</p> <p>heat sinks, 136</p> <p>high-definition. <i>See HD</i></p> <p>high-level synthesis. <i>See HLS</i></p> <p>High-Performance ASIC Prototyping System. <i>See HAPS</i></p> <p>high-speed IO test, 317</p> <p>high-speed TDM. <i>See HSTDMD</i></p> <p>histograms, 73</p> <p>HLS, 39</p> <p>HSTDMD</p> <ul style="list-style-type: none"> multiplexing channels, 263 pin multiplexing, 417 <p>hybrid prototype, 116</p> <p>hybrid verification environment, 364</p> <p>I/O clocks, 51</p> <p>Identify</p> <ul style="list-style-type: none"> signal tracing, 85 <p>Identify tools, 81</p> <p>IICE, 85</p> <p>image processing, 375</p> <p>impact</p> <ul style="list-style-type: none"> board arrangement, 129 connector flexibility, 151 design changes, 111 interconnects, 154 IO limitations, 109 logic placement, 222 map resources, 148 <p>impact analysis, 230</p> <p>Certify tool, 230</p> <p>implementation constraints, 213</p> <p>implementation effort, 110, 111</p> <p>improper inter-FPGA connectivity, 321</p> <p>in-circuit debugging, 77</p> <p>incremental flow</p> <ul style="list-style-type: none"> runtime, 347 <p>Incremental flow</p> <ul style="list-style-type: none"> place & route, 350 <p>incremental flows, 108</p> <ul style="list-style-type: none"> CP boundaries, 348 <p>incremental implementation, 77</p> <p>incremental synthesis, 64</p>
--	---

place & route flow, 351
incremental synthesis flow, 347
inference, 282
 directives, 193
 operator, 302
 SoC element, 189
 test points, 78
inferred soft IP, 301
Innovator
 CHIPit, 386
 HAPS, 386
 system prototype, 385
 VCS, 385
Innovator tool, 6
input and output timing, 323
input clock source selector, 127
instantiated memories, 46
instantiated soft IP, 301
instantiation
 library component, 185
 replace with RTL code, 186
 SoC element replacement, 191
 template, 191
instrumentation
 bus-based, 336
integration use models, 385
inter-clock timing, 324
interconnect
 deferred, 155
 direct, 154
 flexibility, 157
 flight time, 156
 schemes, 156
 topology, 150
interconnect configuration, 147
interconnect delay, 153
interconnect resources, 98
interface
 co-verification, 375
interfaces to external RAM, 196
inter-FPGA connections, 328
inter-FPGA connectivity, 104
inter-FPGA delays, 322
inter-FPGA multiplexing, 251
inter-FPGA timing, 103

internal clock skew and glitches, 325
internal connectivity design, 103
Internal hold time, 322
International Technology Roadmap
 for Semiconductors. *See* ITRS
interrupt service routines, 11
investor status check, 357
IO
 configuration, 52
 core and peripheral, 306
 delay, 52
 FPGA configurations, 98
 logic, 52
IO Block. *See* IOB
IO blocks, 52
IO constraints, 242
IO current, 131
IO delays, 344
IO pins, 52
IO voltage, 131
IO voltage supply to FPGA, 329
IO voltages, 132
IOB, 323
IP
 delivery formats, 290
 encrypted source code, 292
 forms and origins, 290
 model representation, 292
 peripheral, 304
 pre-defined models, 19
 prototyping, 305
 RTL source code, 291
 test chip, 284
 Xilinx, 137
IP from the outside-in, 345
IP instantiation, 293, 301
IP reuse, 401
ISE
 back-end tool, 351
ISE Project Navigator, 75
isolation
 block or wrapper, 282
ISRs, 11
ITRS, 21
Joint Test Action Group. *See* JTAG

<p>JTAG, 335</p> <ul style="list-style-type: none"> configuration, 137 read back, 81 <p>JTAG chain, 56, 335</p> <p>junction</p> <ul style="list-style-type: none"> temperature, 135 <p>latches, 279</p> <ul style="list-style-type: none"> storage element configuration, 44 <p>lateral configuration, 358</p> <p>leaf-cell elements, 188</p> <p>Linux</p> <ul style="list-style-type: none"> EDA tools, 106 <p>local compliance tests, 356</p> <p>logic, 97</p> <ul style="list-style-type: none"> IO, 52 <p>logic block</p> <ul style="list-style-type: none"> clock generate, 176 configurable - CLB, 44 FPGA, 43 <p>logic functions, 43</p> <p>logic mapping, 61</p> <p>logic pruning, 183</p> <p>logic removal</p> <ul style="list-style-type: none"> permanent, 96 temporary, 96 <p>logic replication</p> <ul style="list-style-type: none"> tree pipeline, 248 <p>logic synchronization, 95</p> <p>look-up table. <i>See</i> LUT</p> <p>loopback tests, 414</p> <p>low power, 402</p> <p>low voltage differential signaling). <i>See</i> LVDS</p> <p>LUT, 61</p> <p>LUTs</p> <ul style="list-style-type: none"> logic functions, 43 <p>LVCMOS pins, 329</p> <p>LVDS, 158</p> <p>macro</p> <ul style="list-style-type: none"> implementations, 192 instantiate cell, 210 remove RTL element, 181 <p>main board mounting, 359</p> <p>manage multiple FPGAs, 105</p> <p>manual design partitioning, 105</p>	<p>mapping</p> <ul style="list-style-type: none"> DSP block, 62 logic, 61 memory block, 62 SoC design info FPGA, 61 <p>mapping and utilization reports, 217</p> <p>master modes, 137</p> <p>mechanical configuration, 359</p> <p>memory</p> <ul style="list-style-type: none"> bit-enabled, 209 bit-write enable, 211 <i>block mapping</i>, 62 BlockRAMs, 47 configuration, 44 external resources, 47 FPGA block, 47 high performance access, 340 instantiated, 46 instantiation, 194 IP, 97 logic blocks, 43 self-checking model, 206 single-port, 209 SLICEM, 61 SoC designs, 194 wrappers, 197 <p>memory address generators, 48</p> <p>memory architecture, 195</p> <p>memory blocks, 97</p> <ul style="list-style-type: none"> LX760, 97 <p>memory compatibility, 282</p> <p>memory generator, 194, 195</p> <p>memory model, 206</p> <p>memory modules</p> <ul style="list-style-type: none"> link, 153 <p>memory topologies, 211</p> <p>memory types, 46</p> <p>message channel, 370</p> <p>microprocessor units. <i>See</i> MPUs</p> <p>miniaturization, 15, 401</p> <p>minor modifications, 111</p> <p>mixed-mode clock managers. <i>See</i></p> <ul style="list-style-type: none"> MMCMs, <i>See</i> MMCM <p>MMCM, 98</p> <ul style="list-style-type: none"> clock generation, 49
---	--

MMCMs, 172
moderate modifications, 111
modular design principles, 277
modularity, 117
monitor
 power, 134
 temperature, 134
Moore’s Law, 41, 66, 402
MPUs, 21
multicore
 architectures, 22
 systems, 40
multicore CPU requirements, 22
multicore processing, 402
multi-FPGA
 filter test design, 317
multi-FPGA test designs, 317
multi-mode clock managers. *See*
 MCMM
multiple cores
 sequential software, 22
multiple FPGAs
 startup state issues, 332
multiple modular channels., 277
multiplex
 LVDS, 158
multiplexing, 253
 asynchronous, 261
 clock lines, 51
 design guidelines, 261
partitioning, 253
ratio, 159
schemes, 253
 comparison, 264
single-ended, 262
synchronous, 261
time-domain, 104
timing constraints, 265
using LUTs, 60
multiplexing nets
 criteria, 255
multipliers, 62
multiport memory, 207
mux. *See* multiplexing
netlist

gate-level, 70
partitioning, 70
netlist editing, 186
networking
 application domain, 394
 design team, 118
non-convertible gating logic, 176
 guidelines, 176
non-embedded probing, 80
OCI, 10
OEM, 398
on-board clock synthesis, 127
on-board delays, 126
on-board traces, 124, 233
on-chip
 temperature monitoring, 135
on-chip instrumentation. *See* OCI
operator inference, 302
original equipment manufacturer. *See*
 OEM
packet checker, 415
PAD report, 326
parallel synthesis, 349
parallel-to-serial converter, 53
partitioning
 automated, 239
 automated tools, 239
 automatic, 47
 block size, 225
 clock networks, 245
 coarse, 229
 guidelines, 223
 HDL Bridge, 366
 interconnect board, 156
 iteration, 237
 netlist, 59
 performance, 240
 post-synthesis, 68
 power, 128
 pre-synthesis, 67, 68
 SoC design, 110
partitioning flow, 66
 netlist, 70
partitioning runtime, 109
partitioning strategy

<p>clock domain, 238</p> <p>partitioning tools</p> <ul style="list-style-type: none"> replicate, 233 <p>partner co-development, 356</p> <p>path</p> <ul style="list-style-type: none"> daisy-chained, 147 source-synchronous, 153 star-connection, 147 <p>PCIe, 54</p> <ul style="list-style-type: none"> external peripheral, 309 <p>PCIe blocks, 52</p> <p>PCIe-to-SATA, 309</p> <p>PCIexpress channels, 54</p> <p>performance</p> <ul style="list-style-type: none"> system characteristics, 388 <p>performance analysis, 11</p> <p>performance and accuracy, 26</p> <p>performance estimation, 102</p> <p>peripheral voltage requirements, 329</p> <p>permanent logic removal, 96</p> <p>personality board, 117</p> <p>phase-locked loop. <i>See</i> PLL</p> <p>PHY compatibility, 284</p> <p>physical connectivity., 140</p> <p>physical synthesis, 64, 107</p> <p>physical/mechanical, 96</p> <p>pin</p> <ul style="list-style-type: none"> location constraints, 326 location report, 326 <p>pin locations and properties, 213</p> <p>place & route, 59</p> <ul style="list-style-type: none"> reports, 217 <p>plain text netlist, 309</p> <p>platform</p> <ul style="list-style-type: none"> FPGA, 112 <p>PLL, 125</p> <ul style="list-style-type: none"> clock multiplier, 49 on-board, 126, 247 timing circuits, 139 VCO, 49 <p>plufest, 144</p> <p>PlugFest, 38</p> <p>point of failure</p> <ul style="list-style-type: none"> board connectors, 359 <p>populate black boxes, 239</p>	<p>portability, 282, 357</p> <ul style="list-style-type: none"> advantages, 356 robustness, 139 <p>portable prototype, 355</p> <p>post-synthesis partitioning flow, 68</p> <p>power distribution, 132</p> <ul style="list-style-type: none"> board-level, 131 boards, 132 FPGA design, 132 <p>power modeling, 215</p> <p>power monitoring, 134</p> <p>power monitoring circuits., 132</p> <p>power planes, 133</p> <p>power supply, 129, 360</p> <p>pre-adder, 62</p> <p>pre-defined IP models, 19</p> <p>preliminary timing report, 217</p> <p>probing</p> <ul style="list-style-type: none"> limited pins, 81 non-embedded, 80 real-time signal, 80, 336 test header, 79 test points, 78 tools, 81 <p>processor design</p> <ul style="list-style-type: none"> transistors, 15 <p>programmability, 401</p> <ul style="list-style-type: none"> IO voltages, 132 <p>project generators, 186</p> <p>prototype</p> <ul style="list-style-type: none"> hardware, 16 portable, 355 software, 16 <p>prototype platform, 9</p> <p>prototypes</p> <ul style="list-style-type: none"> FPGA-based, 26 <p>prototyping</p> <ul style="list-style-type: none"> architecture, 10 automotive, 397 board cost, 424 board power, 331 capacity, 13 capacity limitations, 16 check board setup, 331 checklist, 162
---	--

clock tree, 246
coding standards, 275
compatible compatible simulation environment, 273
constraint, 59
criteria, 13
design adaptation, 58
design maturity, 93
design trends, 14
development schedule, 273
enable design flows, 287
external components, 94
feasibility study, 39
FPGA- based, 40
FPGA platform requirements, 143
FPGA resources. *See* future designs, 392
hand-over checklist, 270
hard IP, 308
implementation tools, 59
interconnect configuration, 147
IO pad instantiation, 168
IP, 289, 305
IP as part of SoC, 306
IP reuse, 19
models, 10
network designs, 394
overview, 57
partitioning, 58
peripheral IP guidelines, 304
physical implementation, 41
place and route, 59
power reduction, 215
power-control system, 216
pre-silicon, 8
priorities, 13
procedural guidelines, 268
project schedule, 111
real-time dataflow, 28
revision control, 218
RTL coding standards, 274
runtime, 63
SDK environment, 7
size and form factor, 115
SoC design, 4

SoC elements, 166
software development, 11
standards, 274
stub files, 278
synthesis, 57, 63
system clock, 126
three laws, v
validate SoC design, 36
verification, 12
verification plan, 272
virtual, 6
voltage rails, 331
prototyping board, 143
prototyping boards features, 144
prototyping design guidelines, 288, 408
prototyping IP software validation, 307
prototyping limitation ESL, 39
simulator, 38
prototyping platform hardware, 414
prototyping platforms remote access, 336
prototyping team, 215
prototyping utility, 48
prototype speed checklist, 155
prototyping virtual platform, 10
pruning, 181 downstream logic, 182
signals, 183 upstream, 183
pruning results stubs, 184
PVT, 322
quick-pass flows, 109
RAM. *See* memory
RAM topologies, 211
rate adapter circuits, 306
RCS. *See* revision control system

real-time operating system. *See*
RTOS
real-time signal probing, 336
real-time software, 11
real-world data
 speed, 30
receiver, 53
reduce risk
 excess capacity, 433
 expert advice, 435
 interconnect flexibility, 434
 peripheral de-coupling, 434
regional clocks, 51
regulated power supplies, 138
relax timing constraints, 108
relax timing model, 109
remote access
 virtual ICE, 382
replication
 blocks, 238
 clock domain, 246
 clock functionality, 123
 cost, 9
 logic, 247
 partitioning, 234
 reduce IO, 233
 regression tests, 365
 simplify partitioning, 251
 top level, 246
reset synchronization, 247
return on investment. *See* ROI
reusing boards, 345
revision control
 debug plan, 346
 prototype, 218
 self-documenting code, 274
revision control system, 274
risk
 board connectors, 359
 company standards, 274
 per design, 17
 physical IO, 380
 reusable prototype, 275
 support, 141
risk reduction

FPGA prototyping, 101
ROI, 6
RTL
 debugger, 86
 signal tracing, 85
 track software changes, 272
RTL coding, 268
RTL coding standards, 274
RTL Debugger, 88
RTL file
 soft IP functionality, 301
RTL simulator
 VCS, 366
RTOS, 28
 kernel, 29
runtime
 CPs, 349
 guidelines, 108
 incremental flow, 347
 pre-synthesis partitioning, 68
 synthesis, 63
 test script, 75
SATA, 309
SCE-MI, 370
 simulation, 376
SDC, 214
 design constraints, 214
SDK, 7
SDR
 ISERDES and OSERDES, 52
self-checking wrappers, 204
semiconductor design
 trends, 15
semiconductor trends, 401
sequential boundaries, 241
sequential elements, 65
sequential software
 multiple cores, 22
SERDES blocks, 52
serial
 high-speed IP, 54
 interface, high speed, 191
 transfer clock, 252
 ultra-fast transceivers, 53
serial configuration, 137

serial-to-parallel converter, 53
set/reset, 61
signal integrity, 138, 330
 data rates, 53
signal multiplexing, 158
signal probing
 real-time, 78
signal propagation, 105
 boards, 138
signal tracing
 non real-time, 81
signaling, 95
signals
 assign traces, 235
signs-of-life tests, 316
silicon miniaturization, 401
SIMD, 48
simulation
 acceleration, 383
 host-based, 6
 SCE-MI, 376
 testbenches, 218
single-ended
 input buffers, 51
 IO pin, 52
 multiplexed signals, 257
 outputs, 52
single-ended signaling, 258, 261
single-instruction-multiple-data. *See*
 SIMD
single-pass flow, 70
size and form factor, 115
slave modes, 137
slice
 low power DSP, 48
 Virtex-6, 44
SLICEL, 44
 memory configuration, 44
SLICEM, 44
slow clocking, 287
smartphone, 4
SoC, 1
 block mapping, 62
 block removal, 180
 clock networks, 285

design, 2
memory elements, 62
stubs, 183
SoC design, 99
 constraint files, 213
 FPGA resource estimate, 100
SoC design elements, 166
SoC design modifications, 110
SoC logic
 FPGA mapping, 99
SoC synthesis tool, 302
soft IP, 298
software
 running on chips, 18
software development, 11
 SoC modeling, 28
software development kits. *See* SDK
software drivers, 11
software stack, 28
software validation
 prototype IP, 307
source changes
 library elements, 281
source-synchronous paths, 153
special purpose blocks, 98
speed
 design layout, 157
speed performance
 interconnect schemes, 156
stacked configuration, 358
standard design constraint. *See* SDC
star-connection, 147
start-up synchronization, 250
state annotation, 88
sub-block area, 225
supply of FPGAs, 160
support
 development standards, 141
synchronization
 multi-FPGA, 250
 synchronize block boundaries, 286
 synchronize resets, 286
Synchronous operation
 BlockRAM, 47
SYNCORE, 201

<p>Synplify, 101</p> <p>synthesis</p> <ul style="list-style-type: none"> transfer clock, 259 <p>synthesis, 57</p> <ul style="list-style-type: none"> control branching, 184 fast mode, 64 guidelines, 62 implementation constraints, 213 incremental, 64 mapping, 61 physical, 64 prototyping guidelines, 63 <p>synthesis</p> <ul style="list-style-type: none"> incremental flow, 347 <p>synthesis</p> <ul style="list-style-type: none"> incremental, 347 <p>synthesis constraints</p> <ul style="list-style-type: none"> naming rules, 214 <p>synthesis tools, 60</p> <p>synthesizable RTL</p> <ul style="list-style-type: none"> www.opencores.org, 343 <p>synthetic operators, 302</p> <p>synthetic soft IP, 302</p> <p>System ACE</p> <ul style="list-style-type: none"> Xilinx IP, 137 <p>system interfaces, 14</p> <p>system IO, 380</p> <p>system management circuit, 135</p> <p>system monitor, 55</p> <p>system prototype, 385</p> <p>TAP, 56</p> <p>TCL, 75</p> <p>TDM, 158</p> <p>TDM approach</p> <ul style="list-style-type: none"> signal multiplexing, 158 <p>team organization, 16, 40, 272</p> <p>temperature</p> <ul style="list-style-type: none"> alarm, 135 die junction, 135 monitor and manage, 134 monitor loop, 360 sensor, 135 stress, 136 <p>temperature control, 360</p> <p>temporary logic removal, 96</p>	<p>termination settings, 329</p> <p>terminations, 345</p> <p>test access port. <i>See TAP</i></p> <p>test chip</p> <ul style="list-style-type: none"> IP, 295 limitation, 295 link to FPGA, 297 <p>test chips, 308</p> <p>test design</p> <ul style="list-style-type: none"> filter for multi-FPGAs, 317 guidelines, 316 library, 319 <p>test designs, 316</p> <p>test example</p> <ul style="list-style-type: none"> high-speed, 416 low-speed, 417 <p>test GTX pins, 345</p> <p>test points, 78</p> <p>test real-world data, 33</p> <p>testability, 122</p> <p>testbench</p> <ul style="list-style-type: none"> block-level, 273 design validation, 270 partitioned design, 59 simulation, 218 virtual platform, 379 <p>Texas Instruments, 411</p> <p>three laws of prototyping, v, 406</p> <p>time budgeting</p> <ul style="list-style-type: none"> combinational boundaries, 242 sequential boundaries, 241 <p>time of availability, 13</p> <p>time-critical software, 11</p> <p>TIMESPEC, 76</p> <p>timing calculation, 243</p> <p>timing complexity of TDM, 323</p> <p>timing constraints, 213</p> <ul style="list-style-type: none"> guidelines, 240 <p>timing on multiplexed interconnect, 325</p> <p>timing violations, 321, 322</p> <p>TLM</p> <ul style="list-style-type: none"> transactors, 369 <p>tool control language. <i>See TCL</i></p> <p>tools</p>
--	---

<p>generate replacement memory, 201</p> <p>infer soft macros, 300</p> <p>partitioning, 222</p> <p>trace assignment, 235</p> <ul style="list-style-type: none"> global clock, 246 without EDA tools, 237 <p>trace data</p> <ul style="list-style-type: none"> timing resolution, 82 <p>trace matching, 139</p> <p>track software changes, 272</p> <p>transaction-level models. <i>See</i> TLM</p> <p>transceivers</p> <ul style="list-style-type: none"> gigabit, 53 <p>transfer clock, 252, 253, 255, 258, 260, 261, 262, 325</p> <p>transmitter, 53</p> <p>trends</p> <ul style="list-style-type: none"> semiconductor design, 15 <p>tristate control path, 324</p> <p>tristate signals, 324</p> <p>tuning constraints, 215</p> <p>turnaround time, 14</p> <p>UCF, 75</p> <p>umbilical topologies, 116</p> <p>UMRBus, 337, 375</p> <ul style="list-style-type: none"> CAPIM, 338 IP, 340 memory contents, 340 <p>unintended logic removal, 321</p> <p>Universal Multi Resource bus. <i>See</i> UMRBus</p> <p>universal power format. <i>See</i> UPF</p> <p>universal verification methodology.</p> <ul style="list-style-type: none"> <i>See</i> UVM <p>UPF, 128</p> <p>USB</p> <ul style="list-style-type: none"> drivers, 19 interface, 14 <p>USB OTG, 384</p> <p>user constraints. <i>See</i> UCF</p> <p>utilization levels, 103</p> <p>UVM, 93</p> <p>Value Change Dump. <i>See</i> VCD</p> <p>value links, 14</p>	<p>VCD, 85</p> <p>VCS, 366</p> <ul style="list-style-type: none"> functional verification simulator, 373 <p>verification</p> <ul style="list-style-type: none"> hybrid, 365 interface, 365 interface technologies, 375 reuse, 12 transaction-based, 368 <p>Verification Methodology Manual.</p> <ul style="list-style-type: none"> <i>See</i> VMM <p>VHDL</p> <ul style="list-style-type: none"> global signals, 79 <p>Virtex-6</p> <ul style="list-style-type: none"> arithmetic, 47 blockRam, 196 BlockRam, 47 board utilization, 101 clock distribution resources, 51 clock generation, 49 clock line types, 50 CMT, 49 drive current, 329 DSP blocks, 47 FPBA fabric, 55 FPGA, 195 FPGA regions, 51 GTX, 53 IO current drive, 52 IO pin banks, 51 LX760 FPGAs, 112 memory, 195 PCI Express, 55 serial IO, 53 slice, 44 status information, 56 synthesis architecture, 61 Xilinx, 90 <p>virtual</p> <ul style="list-style-type: none"> prototype, 6 <p>virtual ICE, 381</p> <ul style="list-style-type: none"> remote access, 382 <p>virtual platform, 387</p>
--	---

real-world physical connection, 381	waveforms, 89
RTL, 377	white box, 309
RTL testbench, 380	workstations and licenses, 108
software, 378	wrapper, 197
testbench, 379	block, 282
virtual platforms	wrappers
hardware assisted, 19	instantiate FPGA memory, 198
virtual prototype	instantiated memory, 202
RTL, 377	self-checking, 204
virtual prototypes	Xilinx
advantages, 19	back-end tool, 351
timing, 6	Core Generator, 191
visibility	CORE Generator, 196
logic debug, 335	ISE tool, 351
VMM, 373	XMR, 185
VMM-HAL, 373	CAPIM, 338
voltage and temperature. <i>See</i> PVT	inject signals, 185
voltage controlled oscillator. <i>See</i> VCO	yield, 117
voltage domains, 128	multiple projects, 146
	replicating blocks, 238

About the authors

Doug Amos – Synopsys, Inc.

Doug gained an honors degree in Electrical and Electronic Engineering from the University of Bath, England in 1980.

He did his first programmable logic design in the mid-80's, when FPGAs were still called Logic Cell Arrays. Since then, Doug has designed or supported countless FPGA and ASIC designs either as an independent consultant or working with the leading vendors. Doug became Synplicity's first engineer and Technical Director in Europe (Synplicity was acquired by Synopsys in 2008) and has presented widely on FPGA design and FPGA-based prototyping since that time.

Austin Lesea – Xilinx, Inc.

Austin graduated from UC Berkeley in 1974 with his BS EECS in Electromagnetic (E&M) Theory and in 1975 added an MS EECS in Communications and Information Theory.

He has worked in the telecommunications field for 20 years designing optical, microwave, and copper- based transmission system and developing SONET/SDH GPS-based Timing Systems. For the last ten years at Xilinx, Austin was in the IC Design department for the Virtex product line and for the last two years, he has been working for Xilinx Research Labs, where he is looking beyond the present technology issues.

René Richter – Synopsys, Inc.

René holds an MSEE degree, the Dipl.-Ing. der Elektrotechnik, from the Chemnitz University of Technology in Germany in 1999.

He has worked 11 years in the area of FPGA-based Prototyping, first at ISYTEC, then Pro Design and now at Synopsys; each transition as a result of an acquisition. René managed the development of the CHIPit hardware platforms before moving on to become Director of Applications. During this time he developed co-simulation interfaces and prototyping hardware and has implemented many ASIC designs in FPGA. René has also developed prototyping concepts and solutions for customers and he is one of the inventors of the UMRBus and CHIPit product line.

About Synopsys Press

Synopsys Press offers leading-edge educational publications written by industry experts for the business and technical communities associated with electronic product design. The Business Series offers concise, focused publications, such as The Ten Commandments for Effective Standards and The Synopsys Journal, a quarterly publication for management dedicated to covering the issues facing electronic system designers. The Technical Series publications provide immediately applicable information on technical topics for electronic system designers, with a special focus on proven industry-best practices to enable the mainstream design community to adopt leading-edge technology and methodology. The Technical Series includes the Verification Methodology Manual for Low Power (VMM-LP) and the FPGA-based Prototyping Methodology Manual (FPMM). A hallmark of both Series is the extensive peer review and input process, which leads to trusted, from-the-trenches information. Additional titles are nearing publication in both the Business and Technical series.

In addition to providing up-to-the-minute information for design professionals, Synopsys Press publications serve as textbooks for university courses, including those in the Synopsys University Program:

<http://www.synopsys.com/Community/UniversityProgram>

The Synopsys University program provides full undergraduate and graduate level curricula in electronic design. For more information about Synopsys Press, to contribute feedback on any of our publications, or to submit ideas, please navigate to:

http://www.synopsys.com/synopsys_press