

Vaibbhav Taraate

# Advanced HDL Synthesis and SOC Prototyping

RTL Design Using Verilog



Springer

# Advanced HDL Synthesis and SOC Prototyping

Vaibbhav Taraate

# Advanced HDL Synthesis and SOC Prototyping

RTL Design Using Verilog



Springer

Vaibbhav Taraate  
1 Rupee S T (Semiconductor  
Training @ Rs. 1)  
Pune, Maharashtra, India

ISBN 978-981-10-8775-2      ISBN 978-981-10-8776-9 (eBook)  
<https://doi.org/10.1007/978-981-10-8776-9>

Library of Congress Control Number: 2018958948

© Springer Nature Singapore Pte Ltd. 2019

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

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

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

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd.  
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721,  
Singapore

*Dedicated to my great country Bharat Mata  
and  
To my Master*

# Preface

During this twenty-first century, we are witnessing the miniaturization in the intelligent products. The era of miniaturization should continue for the next few decades. The size of the transistor is almost approaching to atom size of 5 nm. In such a context, the SOC design and prototyping domain have substantially grown with the objective to deliver the intelligent and cost-effective products.

If we look at the domestic market, then the applications of SOC-based design in the areas of wireless, multimedia, processors, controllers, image processing, and the interface protocols have grown up substantially during this decade. This has a real impact on the cost of products due to the competitive nature of the market.

If we try to perceive the technology evolution in the present decade, then we can conclude about the evolutions in the EDA algorithms and the processes to cater to the need of the SOC design and validation. Many EDA vendors like Xilinx, Intel FPGA, Synopsys, Cadence cater to the need of the SOC design. These companies have the sophisticated EDA tool chain and the high-density FPGA board support.

By considering all the above, the manuscript is organized into 16 chapters.

**Chapter 1:** ‘Introduction’: This chapter describes the introduction to SOC design, concept of SOC, SOC design flow, and technology process node and shrinking.

**Chapter 2:** ‘SOC Design’: This chapter discusses the SOC design flow and challenges. The need of SOC prototyping and the challenges in the SOC prototyping are also discussed in this chapter.

**Chapter 3:** ‘RTL Design Guidelines’: This chapter discusses the important guidelines and practical considerations which can be useful during RTL design phase. These guidelines can be tweaking of RTL to improve the design performance or the use of other efficient techniques using Verilog constructs.

**Chapter 4:** ‘RTL Design and Verification’: This chapter discusses the RTL design and verification strategies. This chapter is useful to understand the role of the RTL design and verification engineer and important concepts to achieve the efficient SOC prototype!.

**Chapter 5:** ‘Processor Cores and Architecture Design’: The main objective of this chapter is to develop the thought process of the engineers while sketching the architectures and micro-architectures for the processors. This can be helpful to design the products and new ideas. This chapter is useful to understand the hard IP cores’ use during SOC prototyping.

**Chapter 6:** ‘Buses and Protocols in SOC Designs’: This chapter discusses the few protocols used in the design and their use. This chapter also discusses about the bus architecture and data transfer schemes and techniques. This chapter is useful to understand the basics of I2C, SPI, AHB bus protocols.

**Chapter 7:** ‘Memory and Memory Controllers’: The SDRAM or DDR memory controllers are used extensively in the SOC designs. This chapter discusses the memory controllers and interface techniques with the external memory. The timing constraints for such type of controller are a decisive factor for the overall design and are discussed in this chapter.

**Chapter 8:** ‘DSP Algorithms and Video Processing’: This chapter discusses the DSP algorithms and the role of the design engineer to achieve the desired performance for the DSP designs. This chapter is useful to understand the basics of FIR and IIR filter designs using Verilog and the performance improvement for the design. The video encoder and decoder architectures and micro-architecture to design them using Verilog is also discussed with the practical scenarios.

**Chapter 9:** ‘ASIC and FPGA Synthesis’: This chapter discusses the logical synthesis for ASIC and FPGA designs. During the ASIC prototyping, FPGAs are used and how the ASIC designs can be migrated to FPGA is discussed in this chapter. This chapter focuses on the important RTL design concepts, design portioning, block- and chip-level synthesis to start with. The design constraints used during the synthesis are discussed in this chapter with the practical scenarios. This chapter also focuses on the Synopsys DC commands used during the synthesis. The gated clocks and implementation for ASIC and FPGA are discussed with practical examples and scenarios.

**Chapter 10:** ‘Static Timing Analysis’: This chapter discusses the static timing analysis (STA). The timing paths, maximum frequency calculations, input insertion delay, and output insertion delays are discussed in this chapter with the practical scenarios. The Synopsys PT commands are discussed in this chapter. How to achieve the timing performance to meet the timing constraints is also discussed with the practical scenarios. This chapter is useful for the ASIC and SOC designers to understand the timing in the design and to overcome timing violations in the design. Even this chapter discusses the FPGA timing analysis with practical examples and design scenarios.

**Chapter 11:** ‘SOC Prototyping’: This chapter discusses the FPGA functional blocks with their use. The logic inference using FPGA is discussed with the real-life scenarios. This chapter discusses the prototyping challenges and how to overcome them.

**Chapter 12:** ‘SOC Prototyping Guidelines’: This chapter discusses important design guidelines used during SOC prototyping. The prototyping performance is based on how the design is partitioned into multiple FPGAs. What are IO speed and bandwidth? And how synchronizers are used? This chapter focuses on all these aspects in much more detail using the practical examples and considerations.

**Chapter 13: ‘Design Integration and SOC Synthesis’:** This chapter discusses the SOC synthesis and the design partitioning. The chapter focus of this chapter is to address the important aspects while partitioning the design. The chapter is also useful to understand about the concepts like partitioning, synthesis and STA. How to overcome the partitioning challenges and how to efficiently use the synthesis, place and route and STA tools with an incremental approach to validate the complex SOC designs are also discussed in this chapter!

**Chapter 14: ‘Interconnect Delays and Timing’:** This chapter discusses the high-speed interconnects and their need in the design. This chapter focuses on delay aspects, issue, challenges, and solutions to have the high-speed FPGA prototype using multiple FPGAs. The IO multiplexing, time budgeting, and interconnectivity between FPGAs are described using the practical considerations and design scenarios.

**Chapter 15: ‘SOC Prototyping and Debug Techniques’:** This chapter discusses the important considerations while choosing the target FPGA to validate the SOC designs. This chapter even covers the multiple FPGA designs and considerations, risk, and challenges and how to overcome them. This chapter also covers the Xilinx Zynq-7000 device features and the SOC platform considerations.

**Chapter 16: ‘Testing at the Board Level’:** This chapter discusses the important points while testing the board for the SOC design validation. This chapter covers the debug planning, challenges, board testing for the single FPGA and multiple FPGAs. This chapter can give the understanding of the use of the logic analyzer while testing the SOC design. The inter-FPGA connectivity issues and pin and location constraint issues are also discussed in this chapter.

As stated above, the manuscript is organized to cover the SOC design and prototyping concepts using the high-density FPGAs. The readers will be able to enjoy the manuscript due to the examples and practical scenarios listed in the various chapters.

Pune, India

Vaibbhav Taraate  
Entrepreneur and Mentor

# Acknowledgements

When I started writing the book, *Advanced HDL Synthesis and SOC Prototyping*, the thought in my mind was that this should be helpful to the SOC design engineers and it should cover the concepts in the area of the SOC design. This book originated due to my extensive work in the area of RTL and SOC design.

This book is possible due to the help of many people. I am thankful to all the participants to whom I taught the subject on the RTL design at various multinational corporations. I am thankful to all those entrepreneurs, design/verification engineers, and managers with whom I have worked in the past almost around 16 years.

Especially, I am thankful to my dearest friends for supporting me indirectly and encouraging me to write the book in this area. Their indirect contribution is very much helpful to me and special thanks to them for their good wishes.

I am thankful to my wife, Somi; my son, Siddhesh; and my daughter, Kajal, for supporting me during this period. They have not disturbed me during this period, and this book is the outcome of their help during this period.

Especially, I am thankful to my father, mother, and my spiritual master for their faith and belief in me. Their support has made me stronger!

Finally, I am thankful to the Springer Nature staff, especially Swati Meherishi, Avni, Krati, and Praveenkumar Vijayakumar, for their belief and faith in me.

Special thanks in advance to all the readers and engineers for buying, reading, and enjoying this book!

# Contents

<b>1</b>	<b>Introduction</b>	1
1.1	Moore's Prediction and the Reality	2
1.2	ASIC Designs and Shrinking Process Node	5
1.3	Intel Processor Evolution	7
1.4	ASIC Designs	7
1.4.1	Types of ASIC	9
1.5	ASIC Design Flow	10
1.6	ASIC/SOC Design Challenges and Areas	15
1.7	Important Takeaways and Further Discussions	15
	References	16
<b>2</b>	<b>SOC Design</b>	17
2.1	SOC Designs	17
2.2	SOC Design Flow	19
2.2.1	Design Specifications and System Architecture	19
2.2.2	RTL Design and Functional Verification	20
2.2.3	Synthesis and Timing Verification	21
2.2.4	Physical Design and Verification	21
2.2.5	Prototype and Test	22
2.3	SOC Prototyping and Challenges	22
2.4	Important Takeaways and Further Discussions	24
<b>3</b>	<b>RTL Design Guidelines</b>	25
3.1	RTL Design Guidelines	25
3.2	RTL Design Practical Scenarios	26
3.2.1	Parallel Versus Priority Logic	26
3.2.2	Synopsys full_case Directive	28
3.2.3	Synopsys parallel_case Directive	30
3.2.4	Use of casex	31
3.2.5	Use of casez	32

3.3	Grouping the Terms . . . . .	32
3.4	Tri-State Buses and Logic . . . . .	34
3.5	Incomplete Sensitivity List. . . . .	35
3.6	Sharing of Common Resources . . . . .	36
3.7	Design for Multiple Clock Domain . . . . .	42
3.8	Ordering Temporary Variables . . . . .	43
3.9	Gated Clocks . . . . .	44
3.10	Clock Enables. . . . .	44
3.11	Important Takeaways and Further Discussions . . . . .	50
<b>4</b>	<b>RTL Design and Verification . . . . .</b>	<b>51</b>
4.1	RTL Design Strategy for SOC . . . . .	51
4.2	RTL Verification Strategy for SOC . . . . .	52
4.3	Few Design Scenarios . . . . .	54
4.3.1	Shifting of the Data . . . . .	54
4.3.2	Synchronous Rising and Falling Edge Detection . . . . .	54
4.3.3	Priority Checking . . . . .	54
4.4	State Machines and Optimization . . . . .	57
4.4.1	Moore Machine . . . . .	57
4.4.2	Mealy Machine. . . . .	58
4.4.3	Moore Versus Mealy Machine. . . . .	60
4.5	RTL Design for Complex Designs . . . . .	61
4.6	RTL Design at Top Level . . . . .	61
4.7	Important Takeaways and Further Discussion . . . . .	62
<b>5</b>	<b>Processor Cores and Architecture Design . . . . .</b>	<b>63</b>
5.1	Processor Architectures and Basic Parameters . . . . .	63
5.1.1	Processor and Processor Core . . . . .	63
5.1.2	IO Bandwidth and Clock Rate . . . . .	67
5.1.3	Multitasking and Processor Clock Rate . . . . .	67
5.2	Processor Functionality and the Architecture Design . . . . .	67
5.3	Processor Architecture and Micro-architecture . . . . .	70
5.3.1	Processor Micro-architecture . . . . .	73
5.4	RTL Design and Synthesis Strategies . . . . .	81
5.4.1	Block-Level Design . . . . .	82
5.4.2	Top-Level Design . . . . .	82
5.5	Design Scenarios. . . . .	82
5.5.1	Scenario 1: Instruction Set and ALU Design . . . . .	82
5.5.2	Scenario 2: Data Load and Shifting . . . . .	86
5.5.3	Scenario 3: Parallel Data Load . . . . .	87
5.5.4	Scenario 4: Serial Data Processing. . . . .	87
5.5.5	Scenario 5: Program Counter. . . . .	87
5.5.6	Scenario 6: Register Files . . . . .	87

5.6	Performance Improvement . . . . .	89
5.6.1	How to Tweak the RTL to Improve the Design Performance . . . . .	92
5.7	Use of Processors in SOC Prototyping . . . . .	93
5.8	Important Takeaways and the Further Discussions . . . . .	94
<b>6</b>	<b>Buses and Protocols in SOC Designs . . . . .</b>	<b>97</b>
6.1	Data Transfer Schemes . . . . .	97
6.2	Tri-State Bus . . . . .	99
6.3	Serial Bus Protocols . . . . .	100
6.4	Bus Arbitration . . . . .	104
6.5	Design Scenarios . . . . .	104
6.5.1	Scenario 1: Static Arbitration . . . . .	105
6.5.2	Scenario 2: Bidirectional Data Transfer and Registered IOs . . . . .	105
6.5.3	Scenario 3: UART Transmitter and Receiver Design . . . . .	107
6.6	High-Density FPGA Fabric and Buses . . . . .	107
6.6.1	Xilinx-7 Series Transceivers . . . . .	107
6.6.2	Intel FPGA Transceivers . . . . .	111
6.7	Single Master AHB . . . . .	114
6.8	How This Discussion Is Useful During SOC Prototyping? . . . . .	114
6.9	Important Takeaways and Further Discussions . . . . .	117
	Reference . . . . .	117
<b>7</b>	<b>Memory and Memory Controllers . . . . .</b>	<b>119</b>
7.1	Memory . . . . .	120
7.1.1	Dual-Port Distributed RAM . . . . .	120
7.1.2	Single-Port RAM . . . . .	120
7.1.3	Single-Port RAM (Read First Mode) . . . . .	120
7.1.4	Single-Port RAM (Write First Mode) . . . . .	121
7.1.5	Dual-Port RAM . . . . .	121
7.2	Double Data Rate Memory . . . . .	122
7.3	SRAM Controllers and Timing Constraints . . . . .	122
7.4	SDRAM Controller and Timing Constraints . . . . .	127
7.5	FPGA Design and Memories . . . . .	131
7.6	Memory Controllers . . . . .	134
7.7	How This Discussion Is Helpful in SOC Prototyping? . . . . .	135
7.7.1	Xilinx 7 Series Block RAM . . . . .	135
7.7.2	Stratix 10 Memory Controllers . . . . .	137
7.8	Important Takeaways and Further Discussions . . . . .	139
	References . . . . .	139

<b>8</b>	<b>DSP Algorithms and Video Processing . . . . .</b>	141
8.1	DSP Processors . . . . .	142
8.2	DSP Algorithms and Implementation . . . . .	143
8.2.1	LFSR . . . . .	144
8.3	DSP Processing Environment . . . . .	144
8.4	Architecture for the DSP Algorithms . . . . .	145
8.5	Video Encoders and Decoders . . . . .	148
8.6	How the Discussion Is Helpful in SOC Prototyping? . . . . .	149
8.6.1	Intel FPGA DSP Block . . . . .	150
8.7	Design Scenarios . . . . .	152
8.7.1	The Design of the IIR Filter . . . . .	152
8.7.2	FIR Filter . . . . .	154
8.7.3	Barrel Shifters . . . . .	154
8.8	Important Takeaways and Further Discussions . . . . .	157
	References . . . . .	158
<b>9</b>	<b>ASIC and FPGA Synthesis . . . . .</b>	159
9.1	Design Partitioning . . . . .	159
9.2	RTL Synthesis . . . . .	160
9.3	Design Constraints . . . . .	163
9.4	Synthesis and Constraints . . . . .	163
9.4.1	Chip-Level Synthesis and Constraints . . . . .	166
9.5	Synthesis for SOC Prototype Using FPGA . . . . .	166
9.5.1	How Logic Is Mapped Using CLBs? . . . . .	169
9.5.2	How DSP Blocks Are Mapped? . . . . .	169
9.5.3	How Memory Blocks Are Mapped Inside FPGA? . . . . .	169
9.6	Practical Scenarios During FPGA and ASIC Synthesis . . . . .	170
9.6.1	Gated Clocks and Conversions . . . . .	170
9.6.2	Gated Clock Implementation for ASIC . . . . .	170
9.6.3	Gated Clock Implementation for FPGA . . . . .	171
9.7	Important Takeaways and Further Discussions . . . . .	171
	Reference . . . . .	172
<b>10</b>	<b>Static Timing Analysis . . . . .</b>	173
10.1	Synchronous Circuits and Timing . . . . .	173
10.2	Metastability . . . . .	175
10.3	Metastability and Multiple Clock Domain Designs . . . . .	176
10.4	Timing Analysis . . . . .	177
10.4.1	Dynamic Timing Analysis (DTA) . . . . .	177
10.4.2	Static Timing Analysis (STA) . . . . .	178
10.5	Timing Closure . . . . .	178
10.5.1	STA Important Steps . . . . .	178

<b>10.6</b>	<b>Timing Paths in the Synchronous Design . . . . .</b>	<b>180</b>
10.6.1	Input-to-Register Path . . . . .	181
10.6.2	Register-to-Register Path . . . . .	182
10.6.3	Register-to-Output Path . . . . .	183
10.6.4	Input-to-Output Path . . . . .	184
<b>10.7</b>	<b>What Timing Analyzer Should Perform? . . . . .</b>	<b>184</b>
<b>10.8</b>	<b>Setup Time Analysis . . . . .</b>	<b>184</b>
<b>10.9</b>	<b>Hold Time Analysis . . . . .</b>	<b>188</b>
<b>10.10</b>	<b>Clock Network Latency . . . . .</b>	<b>190</b>
<b>10.11</b>	<b>Generated Clock . . . . .</b>	<b>191</b>
<b>10.12</b>	<b>Clock Muxing and False Paths . . . . .</b>	<b>191</b>
<b>10.13</b>	<b>Clock Gating . . . . .</b>	<b>192</b>
<b>10.14</b>	<b>Multicycle Paths . . . . .</b>	<b>192</b>
<b>10.15</b>	<b>Timing for FPGA Designs . . . . .</b>	<b>193</b>
<b>10.16</b>	<b>Timing Analysis for the FPGA Designs . . . . .</b>	<b>194</b>
<b>10.17</b>	<b>How This Discussion Is Useful During Prototyping? . . . . .</b>	<b>194</b>
<b>10.18</b>	<b>Important Takeaways and Further Discussions . . . . .</b>	<b>195</b>
	<b>Reference . . . . .</b>	<b>196</b>
<b>11</b>	<b>SOC Prototyping . . . . .</b>	<b>197</b>
11.1	SOC Prototyping Using FPGA . . . . .	197
11.2	High-Density FPGA and Prototyping . . . . .	200
11.3	Xilinx 7 Series FPGA . . . . .	202
11.3.1	Xilinx 7 Series CLB Architecture . . . . .	203
11.3.2	Xilinx 7 Series Block RAM . . . . .	204
11.3.3	Xilinx 7 Series DSP . . . . .	206
11.3.4	Xilinx 7 Series Clocking . . . . .	207
11.3.5	Xilinx 7 Series IO . . . . .	207
11.3.6	Xilinx 7 Series Transceivers . . . . .	208
11.3.7	Built-in Monitor . . . . .	209
11.4	Important Takeaways and Further Discussions . . . . .	210
	<b>References . . . . .</b>	<b>210</b>
<b>12</b>	<b>SOC Prototyping Guidelines . . . . .</b>	<b>211</b>
12.1	What Guidelines I Should Follow During SOC Prototyping? . . . . .	212
12.2	RTL Modifications to Have FPGA Equivalent . . . . .	213
12.3	What Care I Should Take During Prototyping? . . . . .	214
12.3.1	Avoid Use of Latches . . . . .	214
12.3.2	Avoid Longer Combinational Paths . . . . .	214
12.3.3	Avoid the Combinational Loops . . . . .	216
12.3.4	Use Wrappers . . . . .	216
12.3.5	Memory Modeling . . . . .	217
12.3.6	Use of Core Generators . . . . .	217

12.3.7	Formal Verification . . . . .	217
12.3.8	Blocks Not Mapping on the FPGA . . . . .	217
12.3.9	Better Architecture Design . . . . .	218
12.3.10	Use Clock Logic at Top Level . . . . .	218
12.3.11	Bottom-Up Approach . . . . .	218
12.4	SOC Prototype Guidelines for Single FPGA Design . . . . .	218
12.4.1	Practical Scenarios and Use of Resources . . . . .	219
12.4.2	Efficient Use of FPGA Resources . . . . .	220
12.4.3	Use of Multiple LUTs in the FPGA Design . . . . .	220
12.5	Prototyping Guidelines for Multiple FPGA Designs . . . . .	221
12.5.1	Interfaces and Connectivity . . . . .	224
12.5.2	Clocking and Speed of the Design . . . . .	224
12.5.3	Clock Generation and Distribution . . . . .	225
12.6	IP Use Guidelines During Prototype . . . . .	226
12.7	Guidelines for Pin Multiplexing . . . . .	226
12.8	IO Multiplexing and Use in Prototype . . . . .	226
12.9	Use of LVDS for High-Speed Serial Data Transfer . . . . .	228
12.10	Use the LVDS to Send Clock on Parallel Line . . . . .	228
12.11	Use the Incremental Flows . . . . .	229
12.12	Important Takeaways and Further Discussions . . . . .	229
	Reference . . . . .	230
<b>13</b>	<b>Design Integration and SOC Synthesis . . . . .</b>	<b>231</b>
13.1	SOC Architecture . . . . .	232
13.2	Design Partitioning . . . . .	232
13.3	Challenges in the Design Partitioning . . . . .	233
13.4	How to Overcome the Partitioning Challenges . . . . .	235
13.4.1	Architecture Level . . . . .	235
13.4.2	Synthesis or Netlist Level . . . . .	237
13.5	Need of the EDA Tools for the Design Partitioning . . . . .	237
13.5.1	Manual Partitioning . . . . .	239
13.5.2	Automatic Partitioning . . . . .	239
13.6	Synthesis for the Better Prototype Outcome . . . . .	241
13.6.1	Fast Synthesis for Initial Resource Estimation . . . . .	241
13.6.2	Incremental Synthesis . . . . .	241
13.7	Constraints and Synthesis for FPGA Designs . . . . .	242
13.8	Important Takeaways and Further Discussion . . . . .	245
	References . . . . .	245
<b>14</b>	<b>Interconnect Delays and Timing . . . . .</b>	<b>247</b>
14.1	Interfaces and Interconnects . . . . .	248
14.2	Interface for High-Speed Data Transfers . . . . .	249
14.3	Interfaces for Multi-FPGA Communication . . . . .	250

14.3.1	Ring-Type Connectivity Between FPGAs . . . . .	250
14.3.2	Star Connectivity . . . . .	251
14.3.3	Mixed Connectivity . . . . .	251
14.4	Deferred Interconnects . . . . .	251
14.5	Onboard Delay Timing . . . . .	253
14.6	What Care We Should Take While Designing the Interface Logic? . . . . .	254
14.7	IO Planning and Constraints . . . . .	255
14.8	IO Multiplexing . . . . .	258
14.8.1	MUX-Based IO Multiplexing . . . . .	258
14.8.2	IO Multiplexing Using SERDES . . . . .	258
14.9	IO Pad Synthesis for FPGA . . . . .	259
14.10	Modern FPGAs IOs and Interfaces . . . . .	260
14.11	How This Discussion Is Helpful During SOC Prototyping? . . . . .	260
14.12	Important Takeaways and Further Discussions . . . . .	262
	References . . . . .	262
<b>15</b>	<b>SOC Prototyping and Debug Techniques . . . . .</b>	<b>263</b>
15.1	SOC Design and Considerations . . . . .	263
15.2	Choosing the Target FPGA . . . . .	265
15.3	SOC Prototyping Platform . . . . .	266
15.4	How to Reduce the Risk in the Prototype? . . . . .	267
15.5	Prototyping Challenges and How to Overcome Them? . . . . .	268
15.6	Multiple FPGA Architecture and Limiting Factors . . . . .	269
15.7	Zynq Prototyping Board Features . . . . .	269
15.7.1	Zynq 7000 Block Diagram . . . . .	270
15.7.2	Zynq 7000 Processing System (PS) . . . . .	271
15.7.3	Zynq 7000 Programmable Logic (PL) . . . . .	272
15.7.4	Zynq 7000 Logic Fabric . . . . .	273
15.7.5	Zynq 7000 Clocks . . . . .	273
15.7.6	Zynq 7000 Memory Map . . . . .	273
15.7.7	Zynq 7000 Device Family . . . . .	274
15.7.8	Zed Board . . . . .	275
15.8	Important Takeaways and Further Discussions . . . . .	275
	Reference . . . . .	276
<b>16</b>	<b>Testing at the Board Level . . . . .</b>	<b>277</b>
16.1	Board Bring-Up and What to Test? . . . . .	277
16.2	Debug Plan and Checklist . . . . .	278
16.2.1	Basic Tests for the FPGA . . . . .	278
16.2.2	Add-On Board Tests . . . . .	279

16.2.3	Test the External Logic Analyzer and FPGA Connectivity . . . . .	279
16.2.4	Multiple FPGA Connectivity and IO Test . . . . .	280
16.2.5	Test for the Multiple FPGA Partitioning . . . . .	280
16.3	What Are Different Issues on the FPGA Boards . . . . .	280
16.4	Testing for the Multiple FPGA Interface . . . . .	280
16.5	Debug Logic and Use of Logic Analyzers . . . . .	283
16.5.1	Probing Using IO Pins . . . . .	283
16.5.2	Use of the Test MUX . . . . .	284
16.5.3	Use of Logic Analyzer: Practical Scenario (To Detect the Data Packet Is Corrupted) . . . . .	284
16.5.4	Oscilloscope to Debug the Design . . . . .	285
16.5.5	Debugging Using ILA Cores . . . . .	286
16.6	System-Level Verification and Debugging . . . . .	287
16.6.1	Hardware–Software Coverification . . . . .	288
16.6.2	Transactors and Transaction-Level Modeling . . . . .	289
16.7	SOC Prototyping Future . . . . .	289
16.8	Important Takeaways and Further Discussions . . . . .	290
	Reference . . . . .	290
	<b>Appendix A: Few Synopsys Commands [1]</b> . . . . .	291
	<b>Appendix B: XILINX-7 Series Family</b> . . . . .	293
	<b>Appendix C: Intel FPGA Stratix 10 Devices</b> . . . . .	297
	<b>Index</b> . . . . .	301

## About the Author

**Vaibbhav Taraate** is Entrepreneur and Mentor at ‘Semiconductor Training @ Rs. 1’. He holds a B.E. (electronics) degree from Shivaji University, Kolhapur (1995), and received a gold medal for standing first in all engineering branches. He completed his M.Tech. (aerospace control and guidance) at the Indian Institute of Technology Bombay (IIT Bombay) in 1999. He has over 15 years of experience in semi-custom ASIC and FPGA designs, primarily using HDL languages such as Verilog and VHDL. He has worked with multinational corporations as a consultant, senior design engineer, and technical manager. His areas of expertise include RTL design using VHDL, RTL design using Verilog, complex FPGA-based design, low-power design, synthesis/optimization, static timing analysis, system design using microprocessors, high-speed VLSI designs, and architecture design of complex SOCs.

# Chapter 1

## Introduction



*The number of transistors incorporated in dense integrated circuit will be doubled in approximately 18 to 24 months.*

Gordon Moore

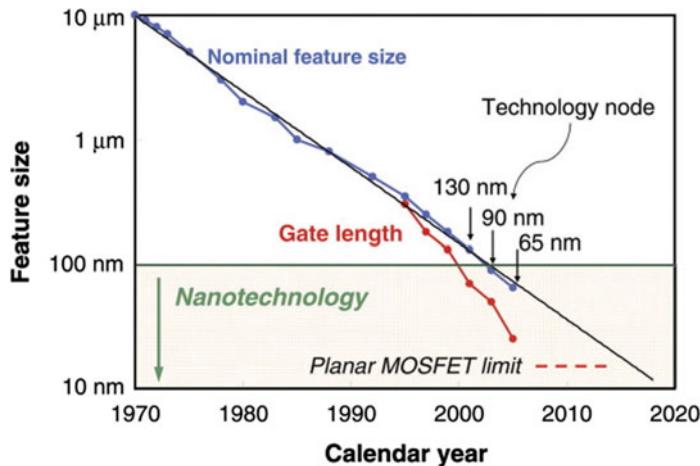
**Abstract** During this decade, the complexity of the ASIC design has increased substantially. The need of the ASICs in the wireless, automotive, medical, and other high processing application has grown. The objective of this chapter is to have discussion about the ASICs and the challenges in the ASIC designs. The chapter even discusses the ASIC design flow, process node evolution, and the SOC architecture. This chapter is useful to understand the steps involved in the design of ASIC.

**Keywords** ASIC · SOC · ASSP · Standard cell · Gate array · Structured ASIC Synthesis · IP · Micro · Multitasking · Clock rate · Data rate · Bandwidth

The ASIC design for the billion gate logic is the need of this decade. The application areas may be wireless communication, high-speed computing, or the video processing. In all these areas, we need to have the high-speed ASIC chips. The prototype for such ASIC or SOC is the requirement to identify the bugs at the implementation level and to measure the performance. In simple words, this avoids the respin of the ASIC chip. In this context, the chapter discusses the ASIC design flow, challenges, and basics of ASIC prototyping.

### 1.1 Moore's Prediction and the Reality

If we consider the introduction of first integrated circuit (IC) by Jack Kilby during year 1958 at Texas Instruments (TI), then nobody had imagined that the integrated circuit (IC) can become so complex during twenty-first century. During 1965–1975, Gordon Moore, Cofounder of Intel, predicted that ‘The number of transistors in



**Fig. 1.1** Feature size versus calendar year

dense integrated circuit will double approximately in 18–24 months.’ We call this as Moore’s law. More than the law, it is treated as prediction and used to plan the integrated circuit design investment and evolution cycle.

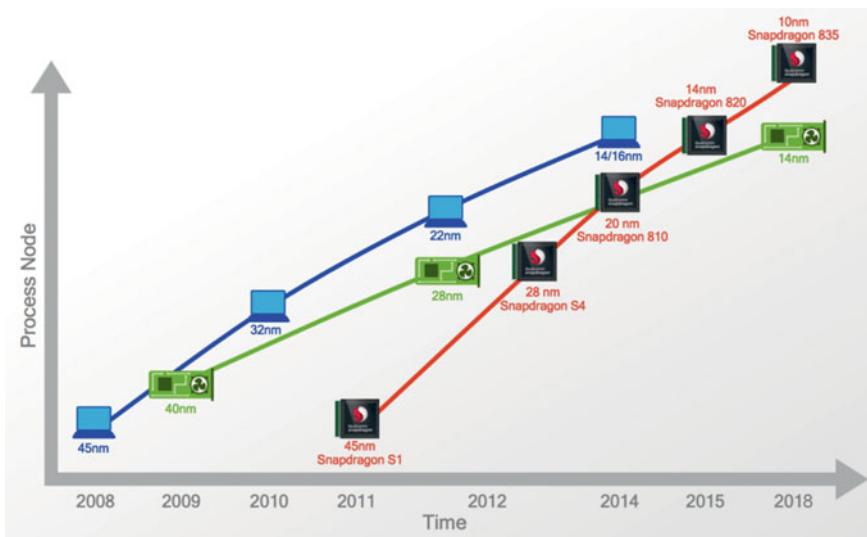
The process node has shrunk enough from few micrometers to 10 nm during last fifty years and even shrinking further. The high-density ASIC designs have many challenges. Those challenges in the current decade are due to the complex design functionality, low-power, and high-speed requirements. Those will be integral part of any design cycle, and those can be overcome by improving the design architectures. Still there are lot many other challenges due to physical and environmental conditions for the lower process node ASICs and SOCs!

If we consider the transistor scaling, then it has some limitations and the real challenge is the device physics. The reality in the design and characterization of the ASIC cell libraries at lower process node is time-consuming phase and involves huge cost. A. S. Rock had stated that ‘The investment requirement for the ASIC chip fabrication doubles approximately 4 years,’ and we call this as Rock’s law or Moore’s second law.

Figure 1.1 gives information about the process node evolution. As shown, the process node has shrunk to almost 10 nm, and even further, it will shrink to 7 nm and below. There will be technology changes and the new manufacturing processes to cope up the challenge of further miniaturization.

According to Intel Technology [1], the transistor density for the 10 nm hyper-scaling provides approximately 2.7 times transistor density improvement as compared to the previous process node of 14 nm.

The limitation in the shrinking is due to the demand and requirement of low-power architectures. Is the shrinking process node meet the required dynamic, static, and leakage power that is one of the questions and even challenges to the designer?



**Fig. 1.2** Mobile application process technology

Let us consider the mobile SOC; the end user needs functionality at the lower cost. So the SOC design challenge in this area is the design of chipset having low power consumption, multitasking and the design functionality, optimization, and so on. As shown in Fig. 1.2, the mobile SOC chipsets are designed using 10/14 nm process node during 2016 and the process node will be evolving further to meet the consumer demands (Fig. 1.2).

International Technology Roadmap for Semiconductors (ITRS) pays much more attentions on the chip-level system design and the design strategies. ITRS assesses the design trends, design technologies, and future development to make the SOC designs more robust. ITRS produces the road map with new additions which can be even applied to the billion gate SOC designs. The major objective of ITRS is to produce roadmap for the ASIC design.

The important points in most of the road maps by ITRS are the cost of the design, manufacturing cycle time, and the target design technology. For the semiconductor customers, the major challenge is the NRE cost which is of millions of dollar. Few of the ITRS road maps produced in the past decade focuses on the reduction in the cost of the design. The important message from such road-maps is that the NRE cost for mask and testing has reached up to few hundred million dollars during this decade and if due to design specification changes or due to major shortfalls in the design if design respins then such costs will multiply. Due to changes in the process technology, the design product life cycle has shortened, and due to that, the time to market is very critical issue for the semiconductor design and manufacturing companies.

If we consider ASIC design, then the design or verification cycle is few months and manufacturing time is few weeks. There are uncertainties in the design and verification but low uncertainty in the chip manufacturing. Under such circumstances, the investments in the process technology have dominated the investments in the design technology. But the important point is that the design cost of the power-efficient ASICs/SOCs during year 2016 is almost around few million dollars versus the hundreds of million dollar investments in the past decade.

Still if we consider ASIC applications, they need software and hardware communication; so during this decade, systems are typically of the embedded type. Almost around 70–80% of the cost is invested to develop software for such systems. During this decade, the ASIC test cost has significantly grown, and for any complex design, the cost of verification is much more as compared to the design cost!

The ITRS assessment and the road map is classified into two major verticals. One is the silicon complexity which is related with the physical design of the chip and other is the system complexity which is related with the system design scenarios and complex functionality.

Most of the ITRS reports suggest the following important highlights related with the physical design:

1. **High-frequency devices and interconnects:** The major challenge is due to the noise, signal integrity, delay variation, and cross-coupling of the devices.
2. **Non-ideal scaling of the parasitic and supply or threshold voltages:** Due to non-ideal scaling, the real challenge is to meet the power constraints.
3. **Interconnect Performance:** How to scale the interconnect performance to establish the communication is one of the challenges.
4. **System-wide clock synchronization:** It is not feasible to implement the synchronous clocking structure for the overall system due to the low power and uniform skew requirements.

Design and manufacturing companies need to think about all these challenges during the design of low-cost, low-power ASIC chips. During this decade, we are witnessing the real limitation in shrinking and doubling of the transistors in dense integrated circuits which has the real impact on the overall road map for the new processor availability!

Apart from these physical design challenges, the system designer needs to think about the cost for the verification and testing, the long verification cycle, the block reuse for the hierarchical designs, hardware and software codesigns, and last but not least the design/verification team size and the geographical locations. In future also, these will remain as important challenges.

The standard node value for year 2017–2020 is shown in the following Fig. 1.3. So we will be able to use the Intel chips of 6.7-nm process node by year 2020.

**Standard Node Value by Year**

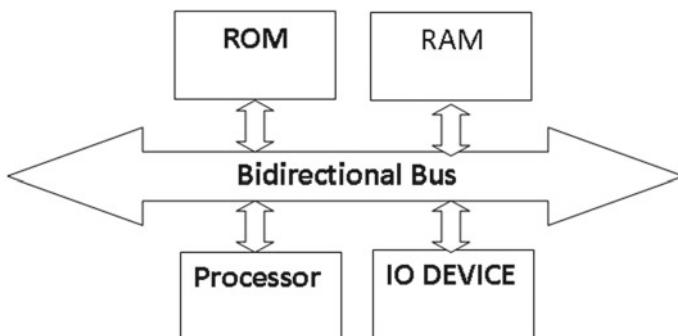
Company	Current	2016	2017	2018	2019	2020
Global Foundries	16.6nm	NA	NA	8.2nm	NA	NA
Intel	<b>13.4nm</b>	NA	9.5nm	NA	NA	6.7nm
Samsung	16.6nm	12.0nm	NA	8.4nm	NA	NA
TSMC	18.3nm	<b>11.3nm</b>	<b>8.2nm</b>	NA	<b>5.4nm</b>	NA

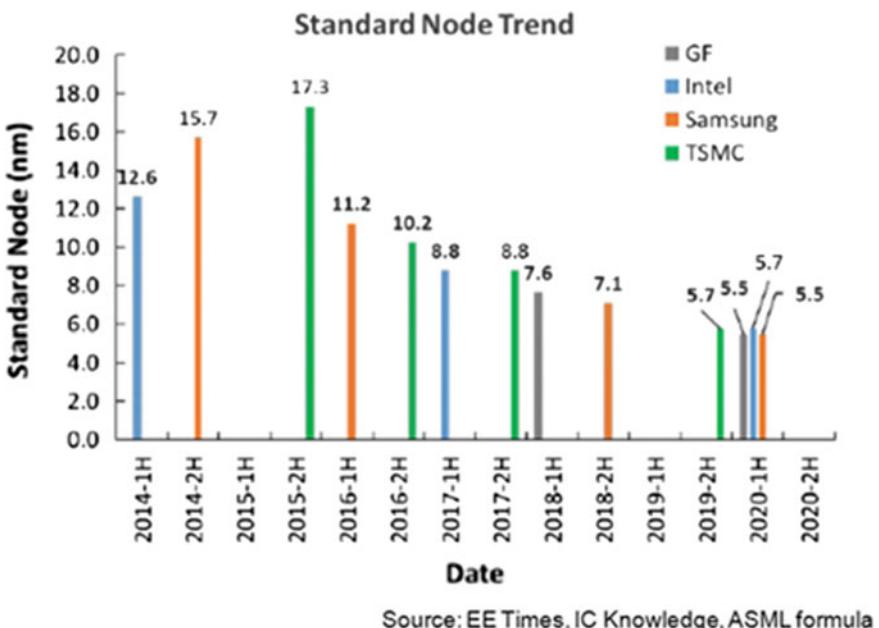
**Fig. 1.3** Standard node year by year

## 1.2 ASIC Designs and Shrinking Process Node

Consider the last century where the chip had single processor and multiple peripheral/memories devices. Consider Fig. 1.4 and as shown, the single processor communicates with the memories (RAM, ROM) and peripheral using the common bus. That was the requirement during the last century when the microprocessors were at the evolving stage.

During this decade, these kinds of SOCs are available in the market at lower cost and can be used in the embedded system design. They can be available by quoting their part numbers. So they are called as ASSP. The traditional application-specific standard product (ASSP) shown has the single processor communicating with the

**Fig. 1.4** Single-processor system



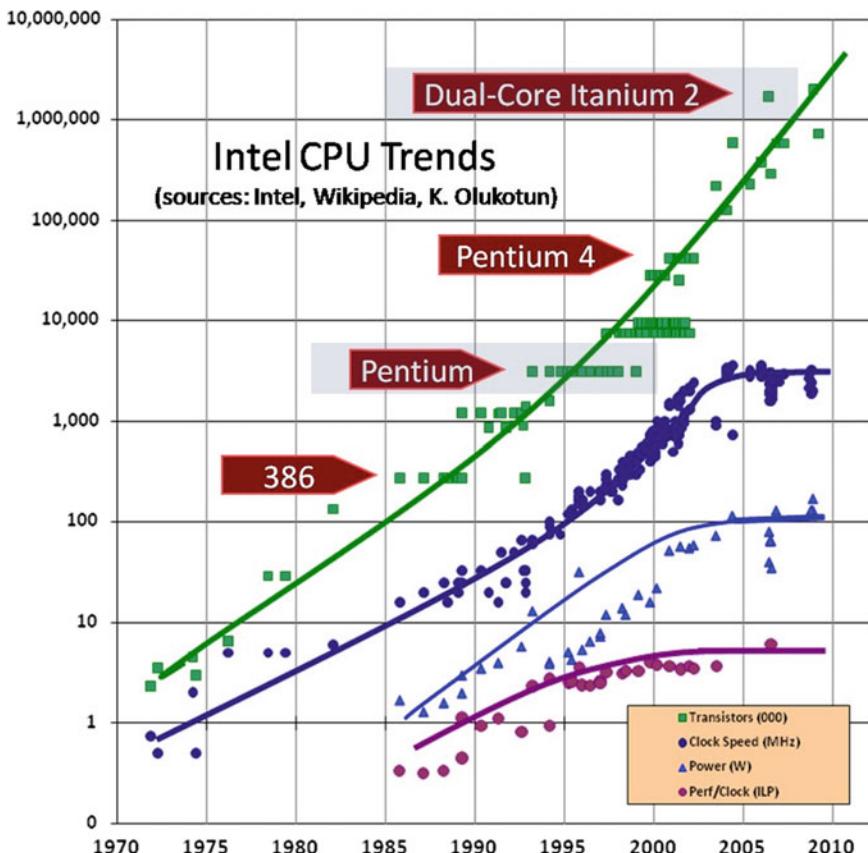
**Fig. 1.5** Standard node trend

IO device, ROM, RAM using shared bus. The gate count of the processor during early 1980s was almost around few thousand logic gates to few lakh logic gates. The design speed was in few MHz, and process node was almost 600  $\mu\text{m}$ . In the present scenario, the process node is 10 nm and the design and manufacturing companies are facing many challenges like the speed and power requirements.

As stated earlier, during this decade, designs are complex and the real requirements for the ASIC designs are extensive parallelism, low-power architectures, embedding the processing functionality including high-end audio/video algorithms on the small silicon area. So the requirement is of the multiple processors and processing engines, reconfigurable environment, and that is the reason of the technology shift toward lower process nodes.

The process node trending during this decade is shown in Fig. 1.5.

So to meet the demand and supply in the market and to innovate the semiconductor products Global Foundries (GF), Intel, Samsung, and Taiwan Semiconductor Manufacturing Corporation (TSMC) can fabricate the ASICs using lower than 7-nm node by the year 2020.



**Fig. 1.6** Intel processor evolution during year 1970–2010

### 1.3 Intel Processor Evolution

As shown in Fig. 1.6, during almost past 50 years, the transistor count has increased exponentially. The clock rate of processors has improved significantly for the required power. This indicates that the real challenge in design of processor chips is to meet the required clocking rate and power requirements.

### 1.4 ASIC Designs

Application-specific integrated circuit (ASIC) is designed for the specific purpose or application. The ASIC chips are designed using full-custom or semi-custom design flow. The full-custom design starts from the scratch, and required cells are designed

for the specific process node. In case of semi-custom ASIC, the prevalidated standard cells and libraries are used and the required additional cells are designed. The additional required functionality may be the design of standard cells and IPs. The ASIC design flow is classified as logical design flow and physical design flow.

The logical design flow involves the design entry using HDL, functional verification of ASIC, synthesis and test insertion and prelayout timing analysis. The physical design flow involves the floorplan, power plan, clock tree synthesis, placement and routing of the design, and finally, the post-layout timing analysis and the testing of the chip.

As stated earlier; in the present decade, the ASICs are complex and may have the billion gates. They can be used in the many applications such as wireless communication, high-speed video processing. In all these applications, the designer needs to have the understanding of the functional specifications, architecture of the design, and even the hardware and software partitioning.

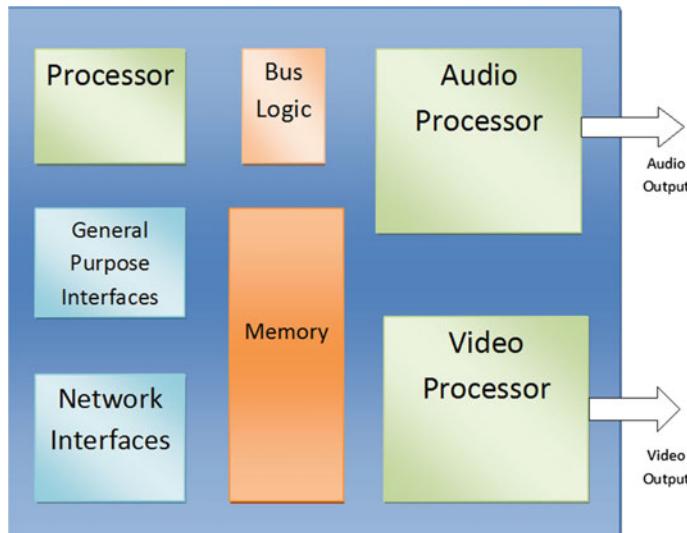
Using the functional specifications, the functionality can be described in the form of the functional blocks; it is also called as architecture of the chip. The complex SOC architecture involves the understanding of the specifications and the creation of the block representation of the design. The architecture design team creates the architecture. The architecture and micro-architecture document involves the functional block details required to realize the ASIC; even the document can have information about the speed, area, and power requirements with the hardware and software partitioning details.

If we consider the evolution of processors during the early 1980s, then the design was very simple due to the need of the single processor and few peripheral devices communicating using the shared bus. During this decade, the designs need the multiple processors, pipelining, concurrency with the architecture exploration for the low power and high speed.

The major challenges in such kind of design are listed below, and mainly they are

- (1) Architecture and system partitioning
- (2) Low-power management
- (3) Use of the functional and timing proven IPs
- (4) Test methodology and equipments
- (5) Verification planning
- (6) Deep submicron effect and integration
- (7) Lesser time to market
- (8) Advance processes and simulation models

The SOC which can be used in the multimedia applications is shown in Fig. 1.7. It has the key blocks like video and audio processing engines, memory, processor, interfaces, bus logic, and general purpose IO interface. The video output and audio output is available from the video and audio processing engine, respectively. The SOC architecture can be improved by adding one or more than one processors in the audio and video processing engine. As each processor performs the operation concurrently, it can produce the high-resolution video and high-quality audio.



**Fig. 1.7** SOC for multimedia

### 1.4.1 Types of ASIC

As stated earlier, the chip designed for the specific purpose or application is called as application-specific integrated circuit (ASIC). Mainly, the complex ASIC consists of the multiple processors with the memories and other functional blocks like the external interface modules. The chip can have analog and digital blocks. For example, consider the design of the ASIC used in wireless communication, and it should have the transmitter and receiver. The chip should have one or more than one processor to perform the parallel processing of the data and should meet the required performance and throughput criteria.

An integrated circuit (IC) is made up of silicon wafer, and each wafer can have thousands of die. Most of the time, we often come across the term which is application-specific standard product (ASSP), and they are available in the market by quoting part numbers. For example, the processor chips, video decoders, audio, and DSP processing chips.

ASICs are primarily classified into following categories, and they are named as:

- Full-custom ASIC
- Semi-custom ASIC
- Gate array ASIC

**Full-custom ASIC:** In such type of ASIC designs, the design needs the characterization of the standard cells. So the design starts with the standard cell design and characterization and validation. The design flow of such design includes the design and validation of the required cells or gates. The preexisting cells are not used in

such kind of ASIC. Consider the design scenario where the design specifications are given to the design team with the requirement of the speed, power, and area. If the preexisting cell does not meet the required performance criteria, then the option is to design the required cells for the target process node. The design cycle in such type of ASIC is longer due to time required to design the standard cells, macros and validation of them.

**Semi-custom ASIC:** In such kind of the design, the preexisting standard cells of logic gates (AND, OR, NOT, EXOR), MUX, flip-flop, and latch are available and used during the design cycle. In this design, team uses the standard cell library where already the cells are predesigned and pretested. This involves the lesser time to market, lesser investments, and even the low risk as compared to full-custom design. Consider the scenario where the standard cell and macros are predesigned and validated for the 10-nm process node. Now, the specifications are given to the design team to design the memory controller using 10-nm process node. In such kind of scenario, the design team uses the predesigned, pretested standard cell libraries. This reduces design cycle time and the risk during design cycle. The standard cell libraries are designed using the full-custom design flow only, and the standard cells can be individually optimized.

**Gate Array ASIC:** The gate array ASICs are further classified as

- Channeled gate array
- Channel-less gate array
- Structured gate array

In the gate array ASIC, the design involves the base array and base cell. The base array is the predefined required pattern of the transistors on the gate array. The base cell is broadly described as the smallest element in the base array. In such kind of ASICs, the cell layout is same for all the cells but interconnects between the cells, and inside of the cell is customized.

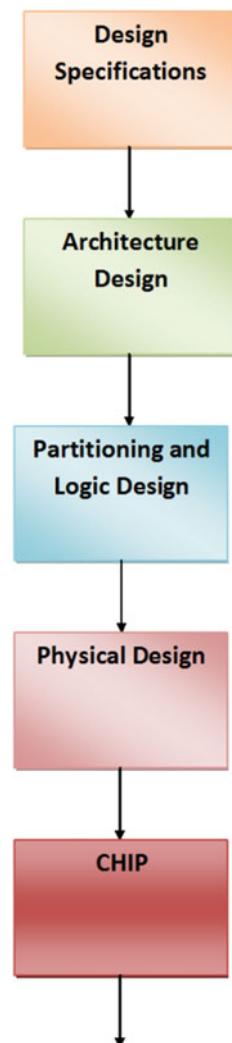
## 1.5 ASIC Design Flow

The ASIC design starts with the idea to realize the product or design for the specific application. The first step is to gather the specifications for the design maybe through the market research or depending on the innovation. The requirement analysis and market survey can be used by the team of architects and engineers to formulate the detailed specifications for the proposed ASIC (Fig. 1.8).

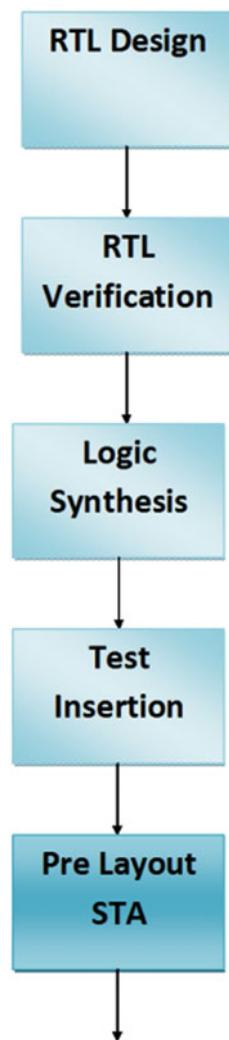
1. **Design Specifications:** The ASIC specifications include the functionality of the design, electrical characteristics, the mechanical assembly. The focus of this book is to design and prototype SOC, and hence, we will elaborate the flow by considering the design specifications.
2. **Architecture Design:** By using the design specifications, the architecture and micro-architecture can be described for the ASIC. The ASIC design is partitioned

into small blocks, and the block-level design is described at the higher level. For example, if ASIC uses processor, then while sketching the architecture the architect team should think about the functionality, speed requirements, external interfaces, pipelining, IO throughput. By using these details, the architecture and micro-architecture for the ASIC can be evolved in the iterative way. Although it is time consuming, the efficient architecture and micro-architecture document is need of the design cycle. This can be used as reference document throughout the design and implementation of ASIC/SOC.

**Fig. 1.8** ASIC design flow



**Fig. 1.9** Logical design flow important steps



3. **Logical Design:** ASIC logical design involves the design partitioning, RTL design, RTL verification, synthesis, test insertion, and the prelayout timing analysis. Figure 1.9 gives information about the ASIC logical design flow at high level.

1. **Specification Understanding** and architectural and micro-architecture for the SOC.

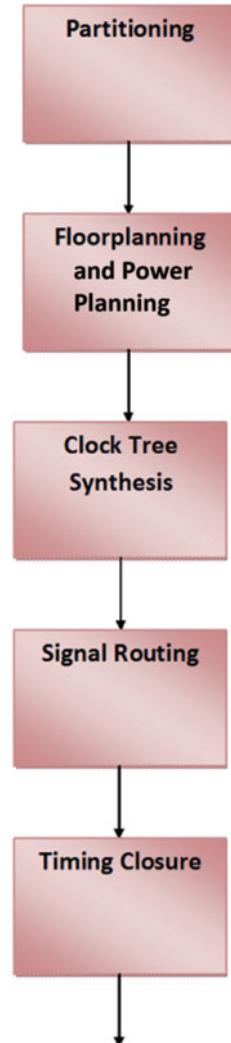
2. **RTL Design:** Design using HDL(VHDL, Verilog, System Verilog).
  3. **Test Insertion:** DFT memory BIST insertion, for designs containing memory elements.
  4. **RTL Verification:** Exhaustive dynamic simulation of the design, to verify the functionality of the design.
  5. **Environment Setting:** This includes the technology library to be used, along with other environmental attributes.
  6. **Design Constraints and Synthesis:** Constraining and synthesizing the design with scan insertion (and optional JTAG) using Design Compiler.
  7. **Block-level STA:** Using Design Compiler's built-in static timing analysis engine.
  8. **Formal Verification:** RTL comparison against the synthesized netlist, using Formality.
  9. **Prelayout STA:** Full chip STA using PrimeTime.
4. **Physical Design Flow:** Figure 1.10 describes few of the important steps in the physical design flow. The flow is iterative depending on meeting the design constraints. If design constraints are met, then the milestone is achieved.

The details of the physical design flow are listed below:

1. **Forward Annotation:** Forward annotation of timing constraints to the layout tool.
2. **Floorplanning:** Initial floorplanning with timing-driven placement of cells, clock tree insertion, and global routing.
3. **Clock Tree:** Transfer of clock tree to the original design (netlist) residing in Design Compiler.
4. **IPO:** In-place optimization (IPO) of the design in Design Compiler.
5. **Formal Verification:** Verification between the synthesized netlist and clock tree inserted netlist, using Formality.
6. **Timing Delay Extraction:** Extraction of estimated timing delays from the layout after the global routing step.
7. **Back Annotation:** Back annotation of estimated timing data from the global routed design, to PrimeTime.
8. **STA:** Static timing analysis using PrimeTime, using the estimated delays extracted after performing global route.
9. **Detailed Routing:** Detailed routing of the design. Extraction of real timing delays from the detailed routed design.
10. **Back Annotate Timing Data:** Back annotation of the real extracted timing data to PrimeTime.
11. **Post-layout STA:** Post-layout static timing analysis using PrimeTime.

12. **Post-layout Simulation:** Functional gate-level simulation of the design with post-layout timing (if required).
13. **Tape Out:** Tape out after LVS and DRC verification.

**Fig. 1.10** Important steps in the physical design flow



## 1.6 ASIC/SOC Design Challenges and Areas

The twenty-first-century ASIC and SOC designs are witnessing the miniaturization challenge as the Moore's law has reached shrinking limitations. The real challenge is to achieve the speed of the ASICs at low power. Nowadays, every human being is interested in having smart phones, intelligent control appliances, and the gadgets. The fun will be during this decade when the massive parallelism in the design of the ASICs and SOCs will try to change the design processes and algorithms. There are many challenges which need to be addressed; few of them are listed below:

1. ASICs can be designed for the high bandwidth and reliable communications to meet the requirements of the end customers.
2. Google-like companies can use the ASICs in the quantum computing systems for the speech recognition
3. Artificial intelligence area will face the challenges due to shrinking process node, and those can be overcome by using the parallelism and parallel processing engines.
4. The medical diagnosis field will consume the large number of ASICs, and new SOCs will be evolved with the parallel processors.
5. Text-to-speech synthesis area will evolve using the parallel processor-based SOCs.
6. The automation in the vehicle controls to give more user-friendly controls to the end user will evolve, and the need of ASICs in the automation will increase drastically.
7. With improved computing and processing power, the SOCs even can be used to control the robots in the hazardous areas with more precision and accuracy,
8. The intelligent sensors, cameras, and scanners to identify the dangerous articles without intervention of human beings can be evolved by using the multi-SOC designs.
9. The automations in the hospitals to monitor the health of the patient from long distance is one of the areas which can evolve using the multiprocessors and ASIC/SOCs.
10. As less area, high speed, and less power are the requirements in all kinds of the ASICs and SOCs, we may witness the technology shift and algorithm evolution to support the massive parallelism during this decade.

## 1.7 Important Takeaways and Further Discussions

As discussed earlier, the following are few important takeaways to conclude the chapter

1. The SOC designs are more complex as compared to ASICs.
2. The ASIC designs can be implemented by using full-custom and semi-custom flow.

3. At the lower process nodes, the real challenge is to achieve the high speed and low power.
4. The modern SOC architecture needs more number of processors, and architecture can be treated as multiprocessor architecture.
5. The concurrency and multitasking can be few of the parameters which need to be considered while designing a system.
6. ITRS road maps' important points are with objective reducing the NRE costs and respinning of ASICs for the future SOC designs.

In the next chapter, we will discuss the SOC designs and the important challenges. The next chapter is also useful to understand about the SOC designs, verification, and prototyping cycle and needs.

## References

1. [www.intel.com](http://www.intel.com)
2. [www.synopsy.com](http://www.synopsy.com)

# Chapter 2

## SOC Design



*Cost of the semiconductor chip fabrication plant doubles every four years.*

Arthur Rock's (Second Moore's Law)

**Abstract** The chapter discusses the basics of SOC design and the SOC design challenges. The SOC design flow and the important steps are discussed in this chapter. The need for SOC prototyping and the challenges in the SOC prototyping are discussed in this chapter. The chapter is useful to prototype engineers to understand the basics of SOC design.

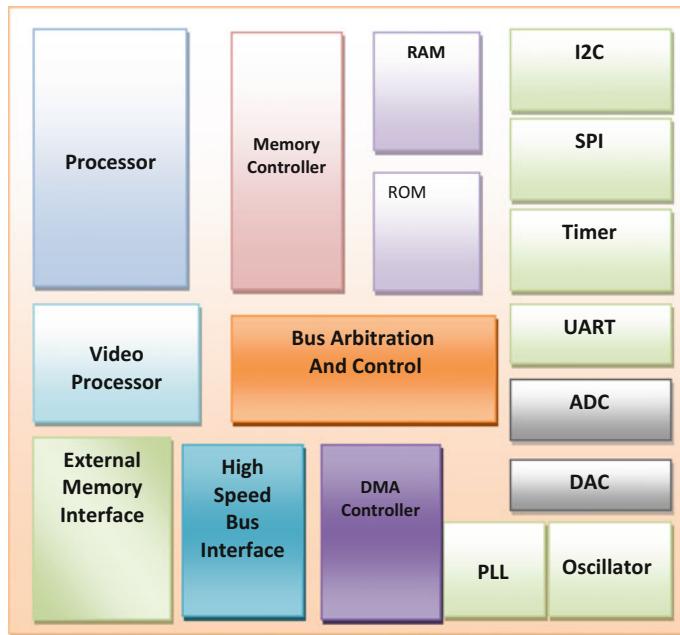
**Keywords** ROM · RAM · Processor · SOC · Bandwidth · IO speed · Clock rate

The basics of the SOC design, SOC design flow, and the prototyping challenges for the SOC designs are discussed in this chapter.

### 2.1 SOC Designs

The design complexity has grown up extensively during this decade. Due to the low power and high speed design requirements in various application the SOC design and prototyping is need of this decade. If we consider any SOC, then the design has analog and digital blocks. Figure 2.1 gives information about few of the SOC components.

1. **Processor and processor core:** The high-density SOCs should have the single or multiple processors. The multiple processor architecture can enable the concurrent execution and parallelism while executing the instructions. In most of the applications, the high-speed, low-power processor architectures are required to perform the complex operations. These operations may be, transfer of the data, floating point operations, audio video processing. Most of the complex



**Fig. 2.1** Complex SOC

SOCs have the general purpose, DSP and video processors and used to improve the overall execution performance of the SOC. Refer chapter 5 for more details about the processor architecture and micro-architecture.

2. **Internal memory:** For internal data storage, the SOC should have memories (RAM, ROM). These memories can be distributed memories or available in the form of the memory blocks. The configurable memory cores can be used to store the large amount of the data. If we consider the DSP processor architecture, then the architecture can be efficient if two separate memories (data and program) can be used. This strategy can be useful to improve the overall architecture performance.
3. **Memory controller:** The DDR or SDRAM controllers can be used to communicate with the external DDR or SDRAM. The high clock rate DDR controllers can be available from the various vendors as the IP. The timing and functional-proven IP use can reduce the design/verification time, and they can be integrated with the SOC components to accomplish the desired tasks. For more details refer chapter 7.
4. **High-speed bus interface:** The high-speed bus interface logic can be used to establish communication with the external host. The protocols and the bus interface logic is elaborated in the chapter 6.
5. **External memory interface:** The application may need flash or SDRAM, and they can be interfaced using the external memory interface.

6. **DMA controllers:** To transfer the large chunk of data, the DMA controllers can be used. The data transfer can be established for the large size of data with high speed.
7. **Serial interfaces:** The serial interfaces like I2C, SPI, and UART can be used to establish communication between the serial devices and the SOC internal components. Refer chapter 6 for more details about the serial interfaces.
8. **ADC and DAC:** The analog devices can be interfaced with the other SOC components using the ADC and DAC.
9. **Clock resources:** The in-built oscillators and PLLs can be used to generate the clocks with the uniform clock skew. The clock distribution network by using multiple PLLs can be used to support the uniform clock skew and the multiple clock domain designs.

The next section discusses the SOC design flow and important milestones.

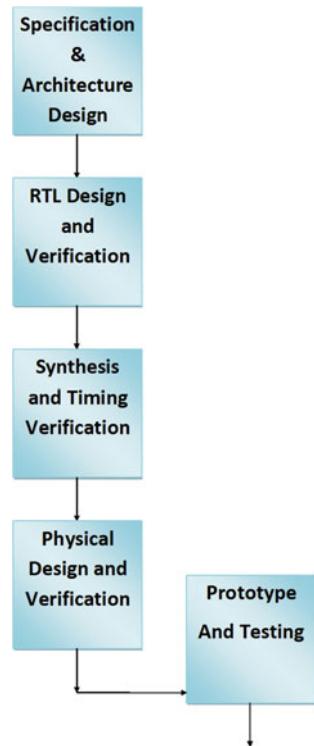
## 2.2 SOC Design Flow

With the evolution of VLSI process technology, the designs are becoming more and more complex, and SOC-based design is feasible in shorter design cycle time due to availability of the prototyping tools. The demand to have product in the shorter design cycle time is possible by using efficient design flow. The design needs to be evolved from specification stage to final layout. The use of EDA tools with the suitable features has made it possible to have the bug-free designs with proven functionality. The design flow is shown in Fig. 2.2 and it consists of the following key milestones.

### 2.2.1 *Design Specifications and System Architecture*

Freezing the design functional specifications for the ASIC or SOC is an important phase. During this phase, the extensive market research is carried out to freeze the functional specifications of the design. Consider the mobile SOC, few important functional specifications can be speed of processor, functional specification of processor, internal memory, display, and its resolution, camera, and resolution of camera, external communication interfaces, etc. More than this, it is essential to have information about the mechanical assembly and other electrical characteristics of the device. They may be power supply and battery charging circuit and safety features. The specifications are used to sketch the top-level floor plan of the chip which we can call as architecture of mobile SOC. Even the important parameters are environmental constraints and the design constraints. The key design constraints are area, speed, and power.

Sketching architecture of any billion gate SOC is one of the difficult tasks as it involves the real imagination and understanding of the interdependability between the

**Fig. 2.2** SOC design flow

hardware and software components. To avoid the overheads on the single processor, the design may need to have multiple processors which can perform the multitasking. The architecture document is always evolved from the design specifications, and it is block-level representation of the overall design. The team of experienced professional can create such type of document, and this can be used as reference to sketch the micro-architecture of the design.

The micro-architecture document is the lower-level abstraction of the architecture documents, and it gives information about the functionality of every block with their interface and timing information. Even this document should give information about the IPs need to be used in the design and their timing and interface details.

The architecture design for SOC and micro-architecture evolution for SOC blocks are discussed in Chaps. 5–8.

### **2.2.2 RTL Design and Functional Verification**

For the complex SOC designs, the micro-architecture document is used as a reference by the design team. The billion gate SOC design is partitioned into multiple

blocks, and the team of hundreds of engineers works to implement the design and to perform the verification. RTL designer uses the recommended design and coding guidelines while implementing the RTL design. An efficient RTL design always plays an important role during implementation cycle. During this, designer describes the block-level and top-level functionality using an efficient Verilog RTL.

After completion of an efficient RTL design phase for the given design specifications, the design functionality is verified by using industry standard simulator. Presynthesis simulation is without any delays, and during this, the focus is to verify the functionality of design. But common practice in the industry is to verify the functionality by writing the testbench. The testbench forces the stimulus of signals to the design and monitors the output from the design. In the present scenario, automation in the verification flow and new verification methodologies have evolved and used to verify the complex design functionality in the shorter span of time using the required number of resources. The role of verification engineer is to test the functional mismatches between the expected output and actual output. If functional mismatch is found during simulation, then it needs to be rectified before moving to the synthesis step. Functional verification is iterative process unless and until design meets the required functionality. For better outcome the team of verification engineers uses the verification plan document. This can result into the better coverage goals.

### ***2.2.3 Synthesis and Timing Verification***

When the functional requirements of the design are met, the next step is synthesis. Synthesis tool uses the Verilog RTL, design constraints, and libraries as inputs and generates the gate-level netlist as an output. Synthesis is iterative process until the design constraints are met. The primary design constraints are area, speed, and power. If the design constraints are not met, then the resynthesis need to be carried out to perform further optimization on the RTL design. After the optimization, if it has observed that the constraints are not met, then it becomes compulsory to modify RTL code or tweak the micro-architecture. The synthesis tool generates the area, speed, and power reports, and gate-level netlist as an output.

The timing verification is carried out by using the gate-level netlist, and this phase is useful to find the presynthesis and post-synthesis simulation mismatches.

The prelayout timing analysis is also important phase to fix the setup violations in the design. The hold violations can be fixed during later stage of the design cycle during post-layout timing analysis.

### ***2.2.4 Physical Design and Verification***

It involves the floor-planning of design, power planning, place and route, clock tree synthesis, post-layout verification, static timing analysis, and generation of GDSII

for an ASIC design. This phase is not discussed in this book. The objective of the remaining chapter is to have discussion on the SOC architecture, micro-architecture, RTL coding, synthesis, and the SOC prototyping using FPGA.

### 2.2.5 *Prototype and Test*

During this phase, the design prototype using FPGA can be validated and tested to understand whether the design meets the required performance, timing, and functionality. This phase is time-consuming milestone, and useful to reduce the overall risks by early detection of bugs. As proof of concept is validated it can be used to avoid respin of the complex ASIC/SOC designs.

## 2.3 SOC Prototyping and Challenges

In the present decade, most of the vendors have powerful FPGA architecture, and the FPGAs are used for the emulation and prototyping. Following are few reasons for the use of modern FPGAs for the prototyping

1. **FPGA architecture:** During emulation and prototyping, the FPGAs can result into the high performance. Nowadays, the FPGAs have the hard processor cores and high-speed interfaces. They can be used efficiently during prototyping.
2. **Testing cost:** For the ASIC the commercial testing is very expensive as compared to FPGA. The high-density FPGA boards can be used to prototype the design and for the emulation.
3. **Verification goals:** Finding out the bugs using simulator can work for the moderate gate count designs, but for the complex designs, the robust verification using application software can be the best choice. This can achieve the desired goals and coverage.
4. **Turnaround time:** The emulation and prototyping phase reduces the overall turnaround time. It reduces the overall risk for the ASIC designs.

As density of SOCs is very high, there are many challenges in the SOC prototyping. Few of the challenges are listed below:

1. **Need of multiple FPGA:** Most of the high-density SOCs needs to be prototyped using multiple FPGAs. The architecture of the FPGA is vendor specific, and even the EDA tool support is vendor specific and may not be effective always. The quality of the partitioning the design into multiple FPGA determines the emulation performance. Another important point is the cost-effectiveness and need of the manpower during the prototyping. Real work needs to be in the area of efficient design partitioning for the better performance using the available FPGA resources and interfaces.

2. **RTL design for ASIC verses FPGA:** The RTL coded for the ASIC does not map easily on the target FPGA. The main reasons are
  - a. There is often difference between the operating frequency of the FPGA and ASIC.
  - b. The clocking architecture and initialization logic is the real bottleneck.
  - c. IO interfaces and memory technology for the ASIC and FPGA may have different architecture. Consider the flash used in the ASIC design, but FPGA uses the DRAM.
  - d. The bus models are different for the ASIC and FPGA. If we compare ASIC verses FPGA, then we can say that no tri-state logic inside FPGA.
  - e. For the ASIC, we need to have the features like debug, controllability, and observability, and they lacks in the FPGA flow.

So during the RTL phase, it is always better practice to code the design for ASIC and to understand the FPGA equivalent of the ASIC designs. During prototyping, the gated clocks, clock, reset trees, and memories need to be mapped into FPGAs by their FPGA equivalent.

3. **Coverification and use of IPs:** The major challenge is the availability of the IPs in suitable form. Most of the time, the IPs are not available in the suitable RTL form. Even to achieve the required speed, it is a requirement that the FPGA interfaces to the simulators or C/C++ models should be design and user friendly, and the availability of such interfaces having the high bandwidth is real bottleneck. Even there is need of the custom interfaces and other communication models for the third-party IPs.
4. **IO bottlenecks:** The emulation speed is limited due to the available IOs and interfaces of the FPGA. The real bottleneck due to IO speed is during the collection of large chunk of data while performing the functional simulation. Even while applying the stimuli, it is essential to consider the speed of IOs and interfaces.
5. **Partitioning:** If the SOCs are partitioned in the better way, then also the communication between the hardware and software using IO interfaces is the real challenge. Bitstream generation while programming the multiple FPGA environments is time-consuming task, and for the recompilation, it may take hours.
6. **In-circuit emulation:** In-circuit or in-environment emulation is one of the challenges. Due to the involvement of other systems in the environment, achieving the real-time performance is the bottleneck if the emulated speed is lesser than the target operational speed. Consider the real practical scenario where Ethernet need to work at speed of 100 Mbps then while prototyping, if the 10 Mbps Ethernet is clocked at 1/10th of the clock rate, then the desired speed can be achieved in the practical system.
7. **Clocking and reset network:** Another challenge is the clock and reset network as they are different in the actual system and emulated system.

## 2.4 Important Takeaways and Further Discussions

1. FPGAs are used extensively during this decade for the prototyping and for the emulation.
2. The emulation using FPGA can be cost-effective and efficient way to test the functionality for the desired performance.
3. The high-end FPGAs from Xilinx and Intel can be used to prototype the SOC as these FPGAs consist of the hard processor cores which operate on higher clock frequency.
4. For SOC design and prototyping, the hardware and software partitioning can play an important role, and the overhead of the communication between the hardware and software can be reduced by using the pipelining and multitasking.
5. The IO interface bandwidth and multitasking features need to be incorporated into the design to achieve the required design performance.
6. The hard processor IP cores can be used during prototyping if the SOC processor core feature matches with the available IP core.

The next chapter focuses on the RTL design guidelines. Few important design guidelines are discussed in the next chapter. The chapter is useful to understand these guidelines and to use them while coding using Verilog.

# Chapter 3

## RTL Design Guidelines



*The first integrated circuit was invented during the year 1958 at Texas Instruments by Jack Kilby.*

**Abstract** The design using Verilog constructs to achieve the better performance should be the objective of the RTL design engineer. The RTL team needs to use the RTL design guidelines while coding for efficient RTL. These guidelines can be tweaking of the RTL to improve the design performance or use of other techniques using Verilog constructs to improve the design performance. This chapter discusses the important guidelines and practical considerations during RTL design.

**Keywords** RTL · Verilog · If-else case always posedge negedge · ASIC synthesis · FPGA synthesis · Multipliers · Pipelining · Multiple clock domain designs · Gray counters · Binary counters · Resource utilization · Resource sharing · Gated clocks · Register balancing · Logic duplication

Use of the design guidelines to improve the performance of the design can help even during implementation stage. Most of the time we observe the need of the RTL tweaks to improve the design performance. The following section discusses about the general guidelines needs to be followed during the RTL design and the role of RTL tweaking using Verilog constructs.

### 3.1 RTL Design Guidelines

Following are the guidelines used during the RTL design cycle:

1. While designing the combinational logic, use the blocking assignments.
2. Use the non-blocking assignments while designing the sequential logic.
3. Do not mix blocking and non-blocking assignments in the same **always** block!

4. Avoid the combinational loops in the design as they are prone to oscillatory behavior.
5. To avoid the simulation and synthesis mismatches use complete sensitivity list by using **always @ (\*)** or using the **always @ (//required inputs, temporary variables)**.
6. Remove the potential unintentional latches by using the **default** while using the **case** construct or by incorporating all the **case** conditions in the **case** constructs.
7. While using the **if-else**, cover all the **else** conditions as missing **else** can infer the latches in the design.
8. If the intention is to design the priority logic, then use the nested **if-else** construct.
9. To infer the parallel logic, use the **case** construct.
10. To avoid the glitches in the design, use the one-hot encoding FSMs.
11. Do not implement the FSM with the combination of the latches and registers.
12. Initialize unused FSM states using reset or by default statements.
13. Use the separate **always** block for the next state, state register, and output logic.
14. For Moore FSM, use **always @ (current\_state)** while coding the RTL for the output logic block and for the mealy machine use the **always @ (current\_state, inputs)**.
15. Do not make the assignments to the same variable or output in the multiple **always** block.
16. Create the separate modules for the functional blocks sensitive to the different clocks.
17. Create the separate module at the top level for the multi-flop level or pulse synchronizer and instantiate them while passing the data between two clock domains.
18. Design the vendor independent RTL by using the inference.

## 3.2 RTL Design Practical Scenarios

The following section discusses the important scenarios during the RTL design and the performance improvement techniques.

### 3.2.1 *Parallel Versus Priority Logic*

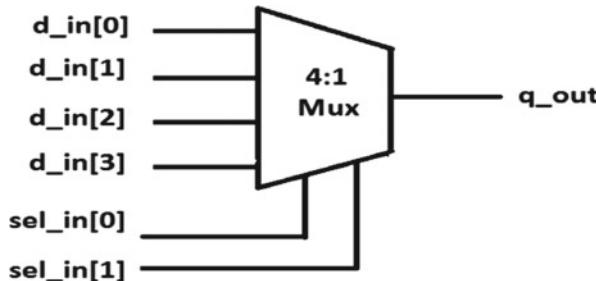
During the RTL design phase, it is important to visualize the synthesis outcome of the RTL. For the moderate gate count ASIC/FPGA functional blocks, it is possible to perceive the resources used for the design.

If the designers have years of experience and have worked on million or billion gate count ASIC, then it is possible to visualize the synthesis outcome of the chip at the higher level. But that is never the objective of the RTL designer.

```
//Verilog code for 4:1 MUX using case

module mux_4to1 (d_in, sel_in, q_out);
    input[3:0] d_in;
    input[1:0] sel_in;
    output q_out;
    reg q_out;
    always@ (*) begin
        case(sel_in)
            2'b00 :q_out = d_in[0];
            2'b01 :q_out = d_in[1];
            2'b10 :q_out = d_in[2];
            2'b11 :q_out = d_in[3];
        endcase
    end
endmodule
```

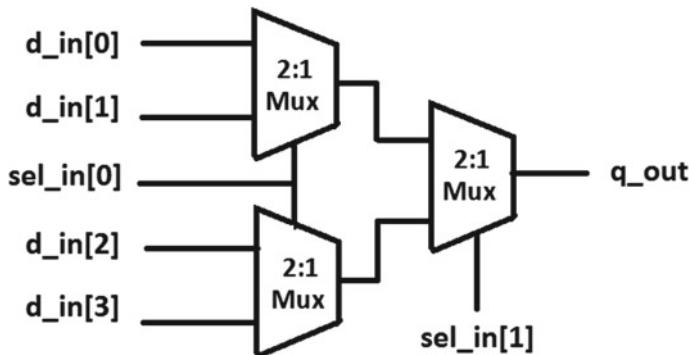
- 1 The blocking assignments are used inside the always block.
- 2 The Verilog blocking assignments are updated in the active queue.
- 3 The blocking assignments are used to design the combinational logic
- 4 The synthesis tool infers the 4:1 MUX with parallel inputs for this example.



### Example 3.1 Parallel combinational logic

Understanding of the logic inference can have added advantages. For example, the parallelism in the design can improve the design performance, or use of the resource sharing can reduce the area although it is specific to the design requirements.

Consider the Verilog code of 4:1 MUX using **case** statement, the **case** construct is used inside the **always** block, and to infer the combinational logic blocking assignments are used. As case construct is used, the output is assigned to one of the inputs depending on the status of select lines. In this, all inputs have same priority. The Verilog code is shown in Example 3.1.



**Fig. 3.1** Synthesis result of 4:1 MUX using case

The synthesis outcome is shown in Fig. 3.1 and as shown it infers 4:1 MUX with four input lines and single output line. The select inputs are used to control the data flow from one of the multiplexer inputs to output.

Most of the times we need to have the priority logic, and under such circumstances the ‘if-else’ statement can be used. As shown in Example 3.2 the 4:1 MUX is described using nested **if-else** statement. Due to use of the **if-else** statement, it infers the priority logic

Synthesis outcome is shown;  $d_{in}[0]$  has highest priority and  $d_{in}[3]$  has lowest priority. The priority logic uses the additional logic to perform the decoding. As shown the decoding logic controls the data transfer through the cascaded chain of 2:1 multiplexer (Example 3.2).

### 3.2.2 *Synopsys full\_case Directive*

Consider the design of the 2:4 decoder having active high enable and active low output. If the design is implemented using the **case** construct and all the **case** conditions are not covered, then the pre-and post-synthesis simulation results differ. Consider the Verilog code Example 3.3.

The `//synopsysfull_case` directive is used (Example 3.4), and then it gives information to the synthesis tool. The directive gives information to the EDA tool as; the **case** statement is fully defined and considers the output assignments for all the unused **case** conditions as don’t care.

While using this directive, care should be taken; the reason being the presynthesis and post-synthesis results may not be matched. The better option is without the uses of this directive, cover all the **case** conditions.

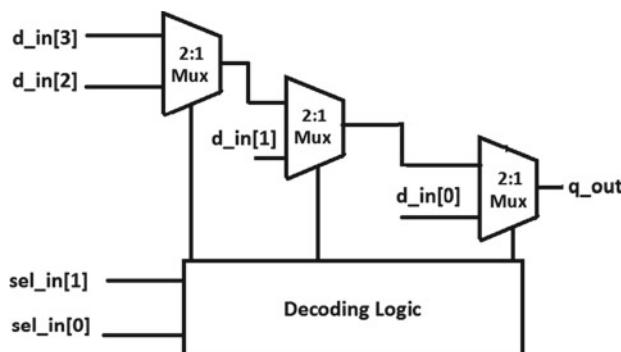
```
//Verilog code for priority 4:1 MUX
module mux_4to1_priority (d_in, sel_in, q_out);

input[3:0] d_in;
input[1:0] sel_in;
output q_out;

reg q_out;

always@ (*)
begin
    if (sel_in==2'b00)
        q_out = d_in[0];
    else if (sel_in==2'b01)
        q_out = d_in [1];
    else if (sel_in==2'b10)
        q_out = d_in [2];
    else
        q_out = d_in [3];
end
endmodule
```

- 1 The blocking assignments are used inside the always block.
- 2 The Verilog blocking assignments are updated in the active queue.
- 3 The blocking assignments are used to design the combinational logic
- 4 The synthesis tool infers the 4:1 MUX with priority logic. The  $d_{in}[0]$  has highest priority and  $d_{in}[3]$  has lowest priority.
- 5 The priority logic is inferred due to nested if-else



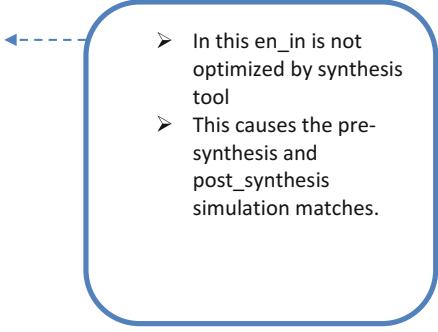
**Example 3.2** Verilog code of priority MUX

```

module decoder-2to4 (y_out, i_in, en_in);

input [I:0] i_in;
input en_in;
output[3:0]y_out;
reg [3:0]y_out;
always @ (*)
begin
    y_out = 4'h1;
    case({en_i,a_in})
        3'b1_00 :y_out = 4'b1110;
        3'b1_01 :y_out = 4'b1101;
        3'b1_10 :y_out = 4'b1011;
        3'b1_11 :y_out = 4'b0111;
    endcase
end
endmodule

```

- 
- In this *en\_in* is not optimized by synthesis tool
  - This causes the pre-synthesis and post\_synthesis simulation matches.

**Example 3.3** Verilog code without full\_case

### 3.2.3 Synopsys parallel\_case Directive

Most of the time we observe the overlapping **case** conditions which can result into the priority logic under such circumstances; it is better to use the //synopsysparallel\_case directive. Consider Example 3.5.

The //synopsys parallel\_case directive is used to give information to the synthesis tool. The directive gives information as; all the **case** conditions should be tested in parallel (Example 3.6).

While using this directive, care should be taken; the reason being most of the time the presynthesis and post-synthesis results may not be matched.

```

module decoder_2to4 (y_out, i_in, en_in);

input [I:0] i_in;
input en_in;
output [3:0] y_out;
reg [3:0] y_out;

always@(*)
begin

    y_out = 4'h1;

    case ({en_in, i_in}) //synopsys_full_case
        3'b1_00 : y_out = 4'b1110;
        3'b1_01 : y_out = 4'b1101;
        3'b1_10 : y_out = 4'b1011;
        3'b1_11 : y_out = 4'b0111;
    endcase
end
endmodule

```

- In this *en\_in* is optimized by synthesis tool and will be dangling  
 ➤ This causes the pre-synthesis and post\_synthesis simulation matches.

**Example 3.4** Verilog code using Synopsys full\_case directive

### 3.2.4 Use of casex

It is recommended not to use the **casex** statement in the RTL coding. Instead of using the **casex**, it is better to use the **casez** statement.

Using **casex** ‘x’ is treated as don’t care. The problem may occur while using the **casex** statement when the input tested by **casex** construct is initialized to unknown state. During the post-synthesis simulation, the ‘x’ is propagated to the gate-level netlist as the condition is tested by the **casex** expression.

Consider the example of 2:4 decoder as shown Example 3.7.

```

module encoder_4to2 (y_out, i_in);
  input [3:0] i_in;
  output [1:0] y_out;
  reg [1:0] y_out;
  always @ ( *)
    begin
      y_out = 2'b00;
      case (i_in)
        4'b1??? : y_out = 2'b11;
        4'b01?? : y_out = 2'b10;
      endcase
    end
  endmodule

```

- In this *en\_in* is not optimized by synthesis tool  
 ➤ This causes the pre-synthesis and post\_synthesis simulation matches.

**Example 3.5** Verilog code without parallel\_case directive

### 3.2.5 Use of casez

It can be used while coding for the priority logic and decoding logic. It is recommended to use the **casez** in the RTL design, but care should be taken for the tri-state initialization (Example 3.8).

## 3.3 Grouping the Terms

To improve the design performance, the grouping can be used. This can be accomplished by using the parenthesis. Consider Example 3.9 shown below. In this example, the  $(a_{in} + b_{in} - c_{in} - d_{in})$  result is assigned to *y\_out*. Without the grouping, the synthesis tool infers the cascaded logic consisting of the arithmetic logic elements.

```

module encoder_4to2 (y_out, i_in);
  input [3:0] i_in;
  output [1:0]y_out;
  reg [1:0]y_out;
  always @ ( *)
    begin
      y_out = 2'b00;
      case (i_in) //synopsis parallel_case
        4'b1??? : y_out = 2'b11;
        4'b01?? : y_out = 2'b10;
      endcase
    end
  endmodule

```

- In this *en\_in* is optimized by synthesis tool and will be dangling  
 ➤ This causes the pre-synthesis and post\_synthesis simulation matches.

### Example 3.6 Verilog code using Synopsys parallel\_case directive

The logic inferred is shown in Fig. 3.2, as shown the logic inferred has three adders and they are connected in cascade. In the simple term, it is priority logic and the delay is  $n^*tpd$ , where  $n$  = number of adders and  $tpd$  = propagation delay of the adder.

The RTL description in Example 3.9 can be modified by the use of parenthesis. The modified code is shown in Example 3.10 and it uses the expression as  $y_{out} = (a_{in} + b_{in}) - (c_{in} + d_{in})$ .

The synthesis result is shown in Fig. 3.3 and it infers the parallel logic due to use of the parenthesis. Due to use of the parenthesis, it infers two adders and one subtractor. The subtraction operation is implemented using 2's complement addition. If the delay of every adder is 1 ns, then the overall propagation delay is 2 ns. This technique is used to improve the design performance.

```

module decoder-2to4 (y_out, i_in, en_in);

[1:0] i_in;
 en_in;
output [3:0] y_out;
reg [3:0] y_out;
always @ ( *)
begin
    y_out = 4'h1;
    casex(en_i,i_in)
        3'b1_00 :y_out = 4'b1110;
        3'b1_01 :y_out = 4'b1101;
        3'b1_I? :y_out = 4'b1011;
    endcase
end
endmodule

```

- If enable input has glitch or the MSB of the *i\_in* has glitches then the output during the pre and post synthesis simulation may be different

**Example 3.7** Verilog code using casex

### 3.4 Tri-State Buses and Logic

The tri-state has three values, logic ‘0’, logic ‘1’, and high impedance ‘z’. The tri-state buses are used in the design to establish communication that is data transfer with other functional blocks. More information about the buses and interfaces are discussed in Chap. 6.

Example 3.11 describes the tri-state logic. It is recommended to use the tri-state logic at the top level in the design. The tri-state is used to avoid the bus contentions. Instead of using the tri-state logic, it is better idea to use the MUX-based logic with the enables.

Figure 3.4 is outcome of the synthesis result for the tri-state logic, and the logic can be used to pass the data when ‘enable\_in’ is equal to logic ‘1’. For logic ‘0’ enable input, the output of tri-state logic is high impedance and it is potential free contact.

```

module decoder-2to4 (y_out, i_in, en_in);
















```

- The problem may occur if one of the inputs is initialized to high impedance state.

**Example 3.8** Verilog code using casez

### 3.5 Incomplete Sensitivity List

The incomplete sensitivity list infers the unintentional latches. The synthesis tool ignores the sensitivity list and infers the combinational logic as XOR gate for Example 3.12.

Consider Example 3.13 in this the required inputs are missing in the sensitivity list and under such circumstances, there is mismatch between the pre- and post-synthesis simulation.

If the sensitivity list is missing, then the **always** block is locked during simulation and it is like infinite looping. The synthesis tool infers the combinational logic XOR gate (Example 3.14).

Better solution to avoid such type of scenarios is; to adapt the use of the coding style described in Example 3.15

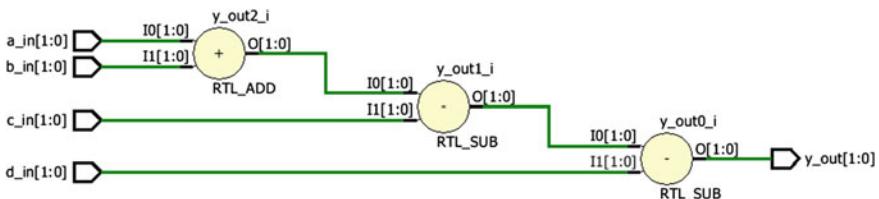
```
// Verilog code without grouping

module logic_without_grouping ( a_in, b_in, c_in, d_in,y_out);
input [1:0] a_in,b_in,c_in,d_in;
output [1:0] y_out;
reg [1:0] y_out;

always@ (*) begin
y_out= a_in + b_in -c_in -d_in;
end
endmodule
```

- ‘always’ block is sensitive to changes on one of the input  
 ➤ On the event on one of the input; ‘y\_out’ is assigned as ‘ $a_{in} + b_{in} - c_{in} - d_{in}$ ’  
 ➤ The design uses the blocking assignment.  
 ➤ This infers the cascaded logic.

**Example 3.9** RTL description without grouping



**Fig. 3.2** Synthesis result for the Verilog code without use of grouping

### 3.6 Sharing of Common Resources

In most of the practical design scenarios, the common resources can be shared by using the fundamental concepts of logic design to achieve area optimization. For

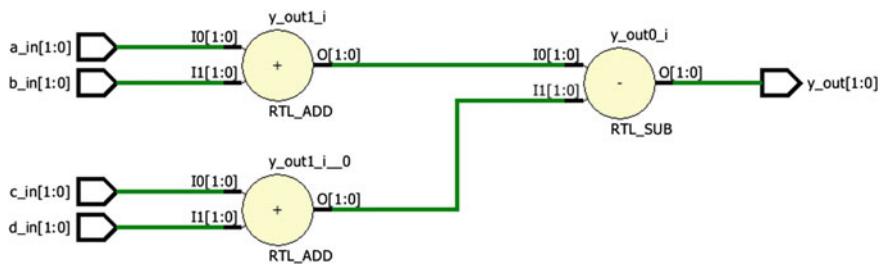
```
// Verilog code with grouping

module logic_with_grouping ( a_in, b_in, c_in, d_in,y_out);
input [1:0] a_in,b_in,c_in,d_in;
output [1:0] y_out;
reg [1:0] y_out;

always@ (*) begin
y_out=(a_in + b_in)-(c_in+d_in);
end
endmodule
```

- The blocking assignments are used inside the always block. Due to grouping the logic infers the parallel adders at the input.
- The result of  $(a_{in} + b_{in}) - (c_{in}+d_{in})$  is assigned to 'y\_out'

**Example 3.10** RTL description using grouping of the terms



**Fig. 3.3** Synthesis result for Verilog code using parenthesis

example, if adders are used and consuming more area, then the area can be reduced by sharing the common adders as resources. This technique plays important role in the improvement of area by optimizing the required gate count (Example 3.16).

```
// Verilog code for the tri state logic

module tri_state (a_in, enable_in, y_out);

input [7:0] a_in,
input enable_in;
output [7:0] y_out ;

reg [7:0] y_out;

always@(*)

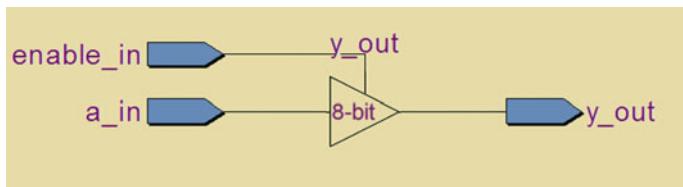
begin

if (enable_in)
    y_out = a_in;
else
    y_out = 8'bz;
end

endmodule
```

- The always block is sensitive to 'enable\_in', 'a\_in'.
- The 'y\_out' is assigned as *a\_in* for enable\_in ='1'
- For enable\_in ='0' y\_out is assigned as high impedance state.

**Example 3.11** Verilog code for tri-state logic



**Fig. 3.4** Synthesis result for the tri-state logic

Instead of using more number of adders, it is better practice and choice to use more number of multiplexers in the design. Consider the Verilog code described in Example 3.16 for the truth Table 3.1. As shown the output needs to be assigned depending on the status of the select input. For 'sel\_in=1', the output 'y\_out' is assigned as '*a\_in+b\_in*' and for the 'sel\_in=0', an output 'y\_out' is assigned as '*c\_in+d\_in*'.

```

module combinational_logic (y_out, a_in,b_in);
  input a_in;
  input b_in;
  output y_out;
  reg y_out;
  always @ ( a_in or b_in)
    begin
      y_out = a_in ^ b_in;
    end
  endmodule

```

- Sensitivity list has all the required inputs  
 ➤ There is no any mismatch between the pre and post synthesis simulation results.

**Example 3.12** Verilog code with complete sensitivity list

```

module combinational_logic (y_out, a_in,b_in);
  input a_in;
  input b_in;
  output y_out;
  reg y_out;
  always @ ( b_in)
    begin
      y_out = a_in ^ b_in;
    end
  endmodule

```

- Sensitivity list has missing input ‘*a\_in*’  
 ➤ There is mismatch between the pre and post synthesis simulation results.

**Example 3.13** Verilog code with the incomplete sensitivity list

The synthesis result for the arithmetic logic without using the concept of resource sharing is shown in Fig. 3.5. As shown in Fig. 3.5, the logic infers two adders and single multiplexer. The adders are used in the data path to perform the addition. The output of multiplexer is controlled by ‘sel\_in’ input, and for the ‘sel\_in’ input as

```

module combinational_logic (y_out, a_in,b_in);
  input a_in;
  input b_in;
  output y_out;
  reg y_out;
  always
    begin
      y_out = a_in ^ b_in;
    end
  endmodule

```

- Sensitivity list is missing  
 ➤ There is mismatch between the pre and post synthesis simulation results.

**Example 3.14** Verilog code with the missing sensitivity list

```

module combinational_logic (y_out, a_in,b_in);
  input a_in;
  input b_in;
  output y_out;
  reg y_out;
  always @(*)
    begin
      y_out = a_in ^ b_in;
    end
  endmodule

```

- *always@(\*)* uses all the required inputs while simulating the design  
 ➤ The pre and post synthesis simulation results the same

**Example 3.15** Verilog code recommended style

logic ‘1’, it generates an output which is addition of ‘*a\_in*’, ‘*b\_in*’. For the logic ‘0’ condition of ‘*sel\_in*’, it generates an output as addition of ‘*c\_in*’, ‘*d\_in*’.

The inferred logic has issue, as both adders are performing operations at the same time; and unnecessary the design has more power dissipation. The result after

```

module resource_sharing (a_in,b_in,c_in,d_in,sel_in,y_out);

a_in,b_in,c_in,d_in;
sel_in;
output [1:0] y_out ;

reg [1:0] y_out;
always @ (a_in, b_in, c_in, d_in, sel_in)
begin
  if(sel_in)
    y_out=a_in + b_in;
  else
    y_out=c_in + d_in;end
endmodule

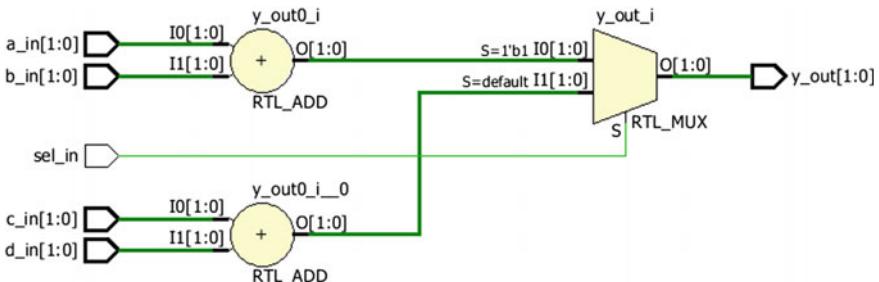
```

- The always block is sensitive to all the required inputs.
- if else is sequential construct and used inside the always.
- For true '*sel\_in*' condition the '*a\_in+b\_in*' is assigned to '*y\_out*'.
- For false '*sel\_in*' condition the '*c\_in+d\_in*' is assigned to '*y\_out*'

**Example 3.16** Verilog code for arithmetic logic without resource sharing

**Table 3.1** Truth table for the arithmetic logic

<i>sel_in</i>	<i>y_out</i>
0	<i>c_in+d_in</i>
1	<i>a_in+b_in</i>



**Fig. 3.5** Synthesis result for the Verilog code without resource sharing

performing the additions waits at the input lines of multiplexers for the active select input, and depending on the status of select line, the output is assigned.

So this kind of technique is less efficient and has more gate count and even has more power dissipation. To overcome this limitation, the resource sharing can be

**Table 3.2** Truth table for the arithmetic logic

sel_in	sig_1	sig_2	y_out
0	c_in	d_in	c_in+d_in
1	a_in	b_in	a_in+b_in

used where the common resources can be shared by pushing the adders forward to the multiplexers. So for this design using resource sharing, more multiplexers are used and less number of adders.

To have efficient resource sharing, push forward the common resources at the output side and use the multiplexers at the input side. Table 3.2 gives information about the strategy used for sharing the common resources.

By modification in the code, the resource sharing can be achieved. The modified Verilog code is described in Example 3.17 and uses the temporary signals as ‘sig\_1’ and ‘sig\_2’.

For logic ‘0’ status on the select line ‘sel\_in’, the ‘sig\_1’ holds the ‘c\_in’ input and ‘sig\_2’ holds the ‘d\_in’ input value. For logic ‘1’ status on the select line ‘sel\_in’, the ‘sig\_1’ holds the ‘a\_in’ input and ‘sig\_2’ holds the ‘b\_in’ input value.

The synthesis result for Example 3.17 is shown in Fig. 3.6

As shown in the figure, the logic is realized by using the single adder and two multiplexers. If the same scenario is considered for the multibit additions, then this type of approach uses lesser area and improves the design performance due to execution of one of the operation at a time.

## 3.7 Design for Multiple Clock Domain

The ASIC designs or design using FPGA can have single or multiple clocks. Most of the time we observe that the single clock domain design does not have the issue of data integrity or data convergence. But if the design has multiple clocks, then the real issue is the data passing from one of the clock domains to another clock domain. To avoid the metastability and the data integrity issues, the data can be passed from clock domain one to clock domain two by using the two-stage or multistage-level synchronizers.

Example 3.18 describes the multiple clock domain design scenario. But in practice, there can be separate design for clock domain one and clock domain two. Instantiate the synchronizer block while passing data between the clock domains.

The synthesis result is shown in Fig. 3.7 and as shown while passing the data from clock domain one to the clock domain two; two-level synchronizer is used. The two-level synchronizer output is valid legal state, although the first flip-flop in the second clock domain goes into the metastable state.

```
module resource_sharing (a_in,b_in,c_in,d_in,sel_in,y_out);
```

```
input [1:0] a_in,b_in,c_in,d_in;
```

```
input sel_in;
```

```
output [1:0] y_out;
```

```
reg [1:0] y_out;
```

```
reg [1:0] sig_1,sig_2;
```

```
always @ ( a_in, b_in, c_in, d_in, sel_in )
```

```
begin
```

```
if (sel_in)
```

```
    begin
```

```
        sig_1 =a_in ;
```

```
        sig_2 =b_in;
```

```
    end
```

```
else
```

```
    begin
```

```
        sig_1 =c_in ;
```

```
        sig_2 =d_in;
```

```
    end
```

```
end
```

```
always @ ( sig_1, sig_2 )
```

```
    begin
```

```
        y_out = sig_1 + sig_2;
```

```
    end
```

```
endmodule
```

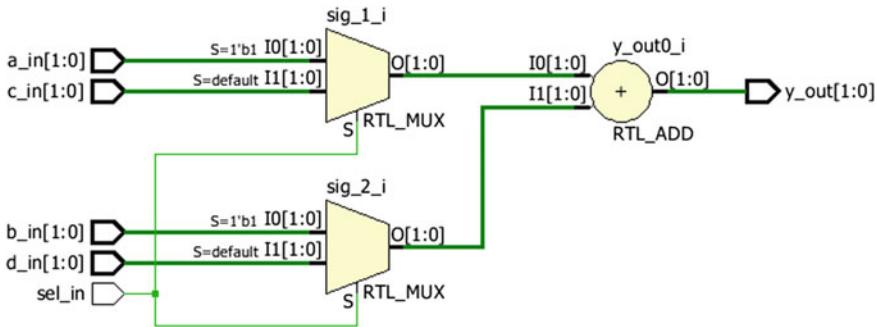
- always block is sensitive to ‘*a\_in*’, ‘*b\_in*’, ‘*c\_in*’, ‘*d\_in*’ and ‘*sel\_in*’.
- if else is sequential statement and used inside the always
- For true ‘*sel\_in*’ condition the input ‘*b\_in*’ is assigned to ‘*sig\_2*’ and input ‘*a\_in*’ is assigned to ‘*sig\_1*’.
- For false ‘*sel\_in*’ condition the input ‘*d\_in*’ is assigned to ‘*sig\_2*’ and input ‘*c\_in*’ is assigned to ‘*sig\_1*’.

- Another always block is sensitive to ‘*sig\_1*’ and ‘*sig\_2*’.
- The blocking assignment is used inside the always block and output ‘*y\_out*’ is assigned to addition of ‘*sig\_1*’, ‘*sig\_2*’

**Example 3.17** Verilog code for the arithmetic logic using resource sharing

### 3.8 Ordering Temporary Variables

During the combinational logic design using **always** block, care should be taken for the assignment of the temporary variable. Consider Example 3.19 in which the statements inside the **always** blocks execute sequentially. As described before assigning, the required new value to the temporary variable the **temp\_reg** is used in the first



**Fig. 3.6** Synthesis result for the Verilog code using resource sharing

assignment. Under such circumstances, the simulator uses the previous latched value for the temp\_reg. This creates the pre-synthesis simulation and post-synthesis simulation mismatches.

The better way to avoid the pre-synthesis and post-synthesis simulation mismatches is by changing the order of the statements inside the always block. This will yield into correct result (Example 3.20).

## 3.9 Gated Clocks

The clock network is hungry net (always toggles) in the design. Due to clock toggling, the design has more dynamic power dissipation. The power dissipation can be reduced by using the clock gating cells. The design using the clock gating concept is described in Example 3.21. The synthesis result is shown in Fig. 3.8.

As shown in the synthesis outcome, the clock of the register is controlled by using the ‘clock\_gate’. The ‘clock\_gate’ signal is generated by using AND logic. But such type of gating strategy is prone to the glitches.

To avoid the glitches, it is recommended to use the clock gating cells. To infer the clock gating, use the vendor-specific EDA tool directives. The ASIC clock gating cells may not be functional equivalent of the FPGA clock gating strategies.

In such kind of scenarios, the tweaking of the RTL is mandatory, or use the gated clock conversion while realizing the design using FPGA. The clock gating conversions and tweaking are discussed in much more detail in Chaps. 9 and 12.

## 3.10 Clock Enables

The sequential design can have the additional enable signal. Depending on the enable signal status, the input data can be transferred to the output. Example 3.22 describes

```
//verilog code for the multiple clock domain design

module multi_clock_design ( a_in,b_in,clk_1,clk_2,y_out);

input a_in , b_in , clk_1, clk_2 ;

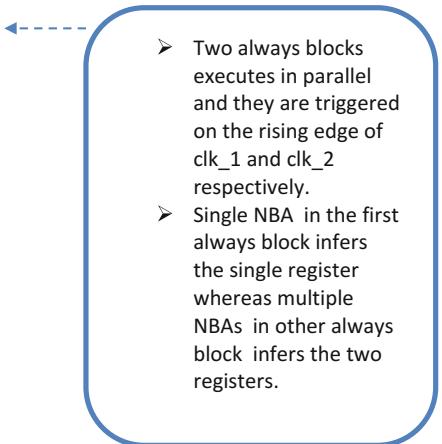
output y_out;

reg y_out;

reg sig_domain_1 , sig_domain_2 ;

always @ (posedge clk_1)
begin
sig_domain_1 <= a_in and b_in;
end

always @ (posedge clk_2)
begin
sig_domain_2 <= sig_domain_1;
y_out<= sig_domain_2;
end
endmodule
```

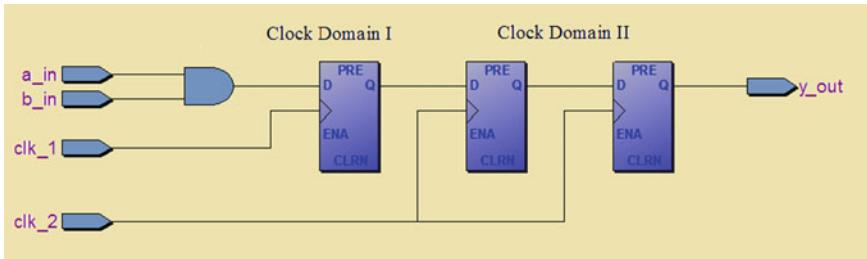
- 
- Two always blocks executes in parallel and they are triggered on the rising edge of clk\_1 and clk\_2 respectively.
  - Single NBA in the first always block infers the single register whereas multiple NBAs in other always block infers the two registers.

**Example 3.18** Verilog code for multiple clock domains

the Verilog RTL having the enable input, and the synthesis result is shown in Fig. 3.9.

As shown in the synthesis outcome, the clock enable is generated and used in the enable path of the flip-flop.

More guidelines related to the practical scenarios, and their use in the practical SOC prototyping is discussed in Chap. 12.



**Fig. 3.7** Synthesis result for the Verilog RTL using multiple clocks

```
module combinational_logic (y_out, a_in,b_in, c_in);
    input a_in;
    input b_in;
    input c_in;
    output y_out;
    reg y_out, tmp_reg;
    always@(*)
        begin
            y_out = (a_in ^ b_in )&tmp_reg;
            tmp_reg = ~c_in;
        end
    endmodule
```

- always block has complete sensitivity list.
- The tmp\_reg is used in the first statement inside always block and it uses the previous value.

**Example 3.19** Verilog code with the improper ordering of temporary variables

```
module combinational_logic ( y_out, a_in,b_in, c_in);  
  
input a_in;  
  
input b_in;  
  
input c_in;  
  
output y_out;  
  
reg y_out, tmp_reg;  
  
always@ (*)  
  
begin  
  
tmp_reg = ~c_in;  
  
y_out = (a_in ^ b_in )&tmp_reg;  
  
end  
  
endmodule
```

- Always block has complete sensitivity list.
- The tmp\_reg is assigned first to not of c\_in and then used in the second assignment.

**Example 3.20** Verilog code with the proper ordering of temporary variables

```

module gated_clock (data_in,clock,load_en,y_out);

input data_in, clk, load_en, clock_en;

output y_out;

reg y_out;

wire clock_gate;

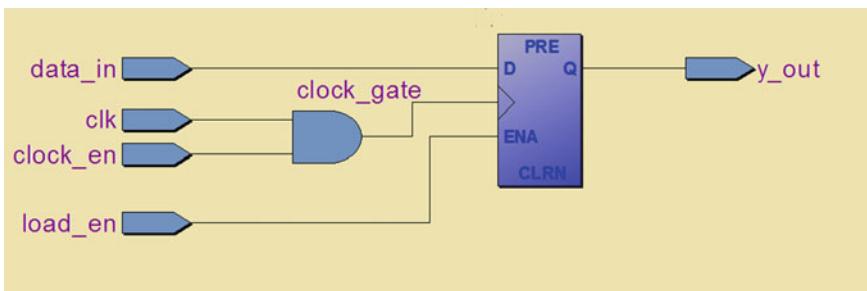
assign clock_gate = (clk and clock_en);

always @ posedge clock_gate
begin
if(load_en)
y_out<= data_in;
end
endmodule

```

- Clock enable  
'clock\_en' signal is used to enable the clock.
- The gated clock  
'clock\_gate' is created by using 'clock\_en' and 'clk'.
- The always block is sensitive to the rising edge of 'clock\_gate'
- For the rising edge of the 'clock\_gate' the 'y\_out' is assigned as 'data\_in' provided that 'load\_en='1'.

**Example 3.21** Verilog code for gated clock



**Fig. 3.8** Synthesis result for Verilog RTL using clock gating

```

module clock_enable (data_in, clk, load_en, clock_en,y_out);
  input data_in;
  input clk;
  input load_en;
  input clock_en;
  output y_out;
  reg y_out;
  reg clock_enable

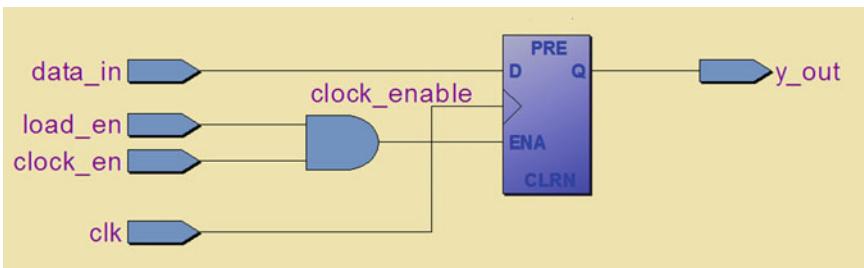
  always @ ( load_en, clock_en)
    begin
      clock_enable=load_en and clock_en;
    end

  always @ (posedge clk)
    begin
      if (clock_enable)
        y_out<= data_in;
      end
    endmodule

```

- The clock enable signal ‘clock\_enable’ is generated by using AND of ‘load\_en’, ‘clock\_en’
- The always block is sensitive to rising edge of clock.
- If ‘clock-enable’ is logic ‘1’ and clk is rising edge then the ‘data\_in’ is passed to the output ‘y\_out’

**Example 3.22** Verilog code using clock enable



**Fig. 3.9** Synthesis result for the Verilog RTL using clock enable

### 3.11 Important Takeaways and Further Discussions

1. While coding the RTL, designer should use the design guidelines.
2. Use the optimization techniques to improve the area, speed, and power.
3. Use the synthesis tool features to optimize the design.
4. To improve the design performance, the design should have the clean and short timing paths.
5. For priority checking, use the nested if-else constructs.
6. Use the case constructs to infer the parallel logic.
7. Use gated clock to reduce the dynamic power.
8. Have RTL with registered inputs and registered output.
9. Use the synchronizers while passing data between clock domains.
10. Use the clock enables to have the clean clock paths.
11. Have clean data and clock paths.
12. Gated clock implementation for the ASIC and FPGA is different, and hence while implementing the prototype using FPGA, use the clock gating conversions.
13. Use tri-state logic at the top level for the design.

This chapter has given us good understanding about the RTL design guidelines using Verilog! The next chapter discusses the RTL design and verification. Even the design and verification strategies for the complex designs are discussed in the next chapter.

# Chapter 4

## RTL Design and Verification



*The design and verification of large-density SOC consumes almost around 80% of the overall product cycle time.*

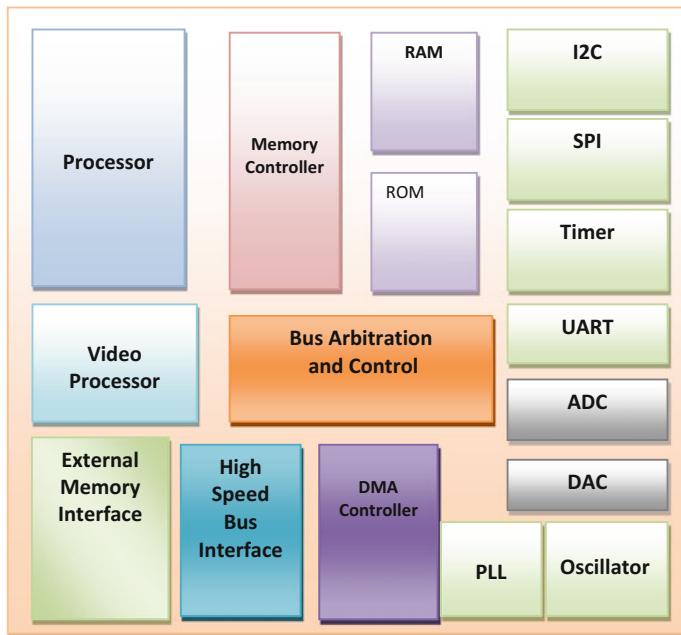
**Abstract** The chapter discusses about RTL design and verification using Verilog. The RTL design and verification strategies are also discussed in this chapter. The chapter even discusses about the FSM performance improvement strategies. The chapter is useful to understand the role of the RTL design and verification engineer and important concepts.

**Keywords** RTL · Verilog · Shift register · Edge detector · Priority checking · Moore machine · Mealy machine · Performance improvement · Verification Coverage · Verification plan · Test case · Corner case

### 4.1 RTL Design Strategy for SOC

For the complex SOC designs, the RTL design phase can consume almost around 10–15% of the design time. The design can be partitioned into the multiple functional blocks, and by using the divide and conquer method, RTL can be coded. For such type of the design, the RTL can be overall integration of the functional-proven IPs, glue, and associated logic with the test and debug logic. The care should be taken by the RTL team to use the following general guidelines as references:

1. Use reference document as the architecture and micro-architecture of the SOC.
2. Understand the functional block interdependability with the other design blocks.
3. As a team member, try to have the clarity of the functional dependencies and the external interfaces.
4. If you are the owner of the functional block and need to code the design using Verilog, then use the RTL design guidelines.



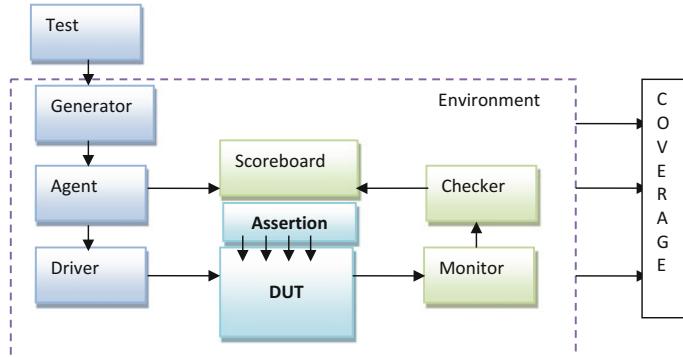
**Fig. 4.1** Complex SOC design

5. As a owner of the RTL functional block, try to refer the micro-architecture to understand the functionality and implement the small-block-level designs using Verilog.
6. Perform the basic verification for each sub-block to validate the functionality of the design.
7. Integrate the sub-blocks at the top and use the test, debug, and tri-state at the top level (Fig. 4.1).

## 4.2 RTL Verification Strategy for SOC

The goal is to verify the functional correctness of the design. For the small gate count design, the basic testbench using Verilog can be used to report the bugs in the design. For the complex designs, the process needs to be automated using the HVLs and the self-checking layered testbenches. Depending on the use of the protocols, the complexity of the layers in the testbenches can be increased to reach the coverage goals.

The RTL verification for the complex design can consume almost around 65–70% of the overall design cycle time, and the following can be done to achieve the coverage goals.



**Fig. 4.2** Layered testbench

1. Have verification plan in place and kick-start verification concurrently with the RTL design phase.
2. Have understanding of the block-level functionality to get the corner cases.
3. Create the test cases and randomize them to carry out the verification for the block-level design.
4. Have the testbench architecture using driver, monitor, and scoreboards and develop the automated sophisticated testbench.
5. Define the coverage goals such as functional, code, toggle, and constrained randomized coverage at the block level and at the chip level.

The testbench should perform the following:

1. Generate stimulus
2. Apply stimulus to the DUT
3. Capture the response
4. Check for the functional correctness
5. Track and measure progress against the overall verification goals

#### What need to be thought about the design inputs while randomizing?

1. Device configuration
2. Environment configuration
3. Input data
4. Protocol exceptions
5. Delays
6. Errors and violations

The layered testbench architecture is shown in Fig. 4.2.

The command layer has the driver which drives the command to the DUT, and the monitor captures the transition of the signals and groups them together in the form of the command. Consider bus write or read command in AHB. The assertions also drive the DUT.

Above the command layer, the functional layer in which the agent or transactor drives the driver after receiving the high-level transaction such as DMA read and write. Such type of transaction can be broken into multiple commands to drive the driver.

To predict the result of the transactions, these commands are sent to the scoreboard and the checker compares the commands from monitor with the scoreboard.

If we consider the H.264 encoder then to test for the multiple frame processing, frame size, and type, these parameters can be configured by using the constrained random values of these parameters. This is what we call as creating the scenario to verify the particular functionality.

## 4.3 Few Design Scenarios

For moderate gate count designs, the RTL description is discussed in this section.

### 4.3.1 *Shifting of the Data*

The Verilog code of the serial-input and serial-output shift register is shown in Example 4.1. As shown, the non-blocking assignments (NBAs) are used inside always block. NBAs are used while describing the sequential logic design. The synthesis outcome is serial-input and serial-output shift register having rising edge of clock and active low asynchronous reset.

### 4.3.2 *Synchronous Rising and Falling Edge Detection*

Most of the time, we need to have the logic to detect the positive edge or negative edge. The synchronous logic which operates on the clock edge and described in Example 4.2 (Fig. 4.3).

### 4.3.3 *Priority Checking*

If most of the level-sensitive signals are arriving at a time and need to be sensed and processed depending on the priority, then the priority encoders can be used. Consider a practical scenario of the processor logic having four-level-sensitive interrupts and priority need to be scheduled for them, and then the logic can be designed by using nested if-else construct of Verilog.

```
//Verilog RTL for the serial input serial output shift register

module shift_register (d_in, clk, reset_n, q_out);

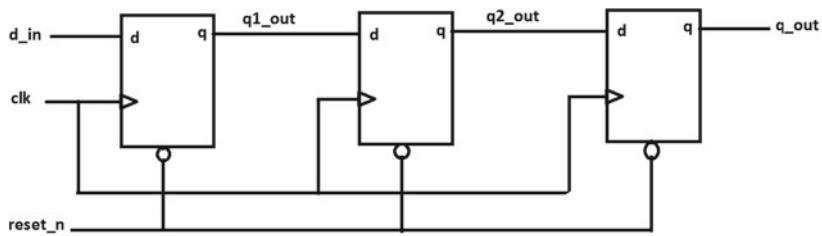
input d_in;
input clk;
input reset_n;
output q_out;

reg q_out;
reg q1_out, q2_out;

always @ (posedge clk or negedge reset_n)

begin
if (~reset_n)
begin
    q1_out <= 1'b0;
    q2_out <= 1'b0;
    q_out <= 1'b0;
end
else
begin
    q1_out <= d_in;
    q2_out <= q1_out;
    q_out <= q2_out;
end
end
endmodule
```

- 1 The non blocking assignments are used inside the always block.
- 2 Due to use of non blocking assignments the logic inferred is serial input and serial output shift register.
- 3 The synthesis tool infers the sequential logic with asynchronous reset.
- 4 The logic inferred has three flip flops.



**Example 4.1** Verilog code of serial-input and serial-output shift register

```
//Synthesizable Verilog for the rising /falling edge detection

module edge_detection (d_in, clk, reset_n, q_out);

input d_in;
input clk;
input reset_n;
output q_out;

reg tmp_q_out;

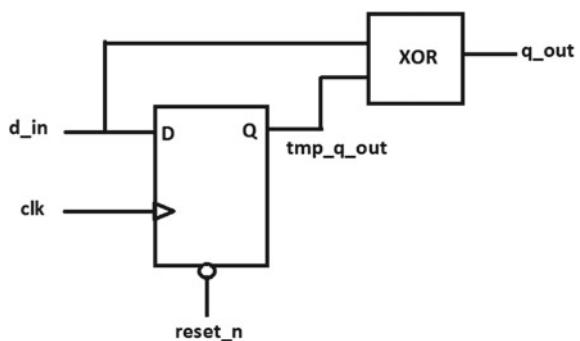
always @ (posedge clk or negedge reset_n)
begin
    if(~reset_n)
        tmp_q_out <= 1'b0;
    else
        tmp_q_out <= d_in;
end

assign q_out = tmp_q_out ^ d_in;

endmodule
```

**Example 4.2** Verilog code for the rising and falling edge detector

**Fig. 4.3** Synthesis result of edge detector



**Table 4.1** Truth table of 4:2 priority encoder

INT3	INT2	INT1	INT0	y1_out	y0_out
1	X	X	X	1	1
0	1	X	X	1	0
0	0	1	X	0	1
0	0	0	1	0	0
0	0	0	0	0	0

Consider the Table 4.1, in which the input INT3 has highest priority and INT0 has lowest priority. Outputs of encoder are y1\_out and y0\_out (Table 4.1).

As shown in the above table, the level-sensitive input INT3 has highest priority and INT0 has lowest priority. If the above table entries are observed carefully, then we can see the output ‘00’ for two input sequences ‘0001’ and ‘0000’. If the output of the encoder is given as input to another functional block, then it is very difficult to understand whether an output of encoder stage is ‘00’ due to input ‘0001’ or due to input sequence ‘0000’. Use the flag\_out as additional output of encoder to detect all inputs are equal to ‘0000’. If all the encoder inputs are having logic zero value, then force flag\_out to logic 1 otherwise flag\_out should be logic 0.

The Verilog code is shown in Example 4.3 and its synthesis result is shown in Fig. 4.4.

## 4.4 State Machines and Optimization

We need to have the finite state machine (FSM) controllers in the design to get the better timing performance. The FSMs are used to implement the controllers, and even they are used to implement the arbitrary counters and sequence detectors. The performance of the FSMs is one of the important aspects to achieve the desired performance of the overall design. The FSMs are of two types—Moore and Mealy.

### 4.4.1 Moore Machine

In the Moore machine, the output is function of the current state only. It needs to wait for one clock cycle to change the output after input change (Fig. 4.5).

```
//Synthesizable Verilog for 4:2 priority encoder

module priority_encoder (INT, y_out, flag_out);

input [3:0] INT;
output [1:0] y_out;
output flag_out;

reg flag_out;
reg [1:0] y_out;

always @ (*)

begin
    y_out = 2'b00;
    flag_out = 1'b1;
    if(INT[3]) begin
        y_out = 2'b11; flag_out = 1'b0; end
    else if(INT[2]) begin
        y_out = 2'b10; flag_out = 1'b0; end
    else if(INT[1]) begin
        y_out = 2'b01; flag_out = 1'b0; end
    else begin
        y_out = 2'b00; flag_out = 1'b0; end
    end
end

endmodule
```

**Example 4.3** Verilog code for 4:2 priority encoder

#### 4.4.2 Mealy Machine

In the Mealy machine, the output is function of both the present input and current state. Output changes immediately in the same clock cycle for the change in the input (Fig. 4.6).

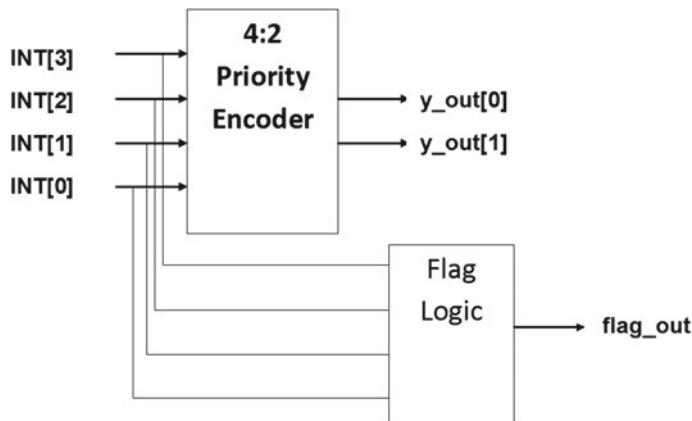


Fig. 4.4 Synthesis result of 4:2 priority encoder

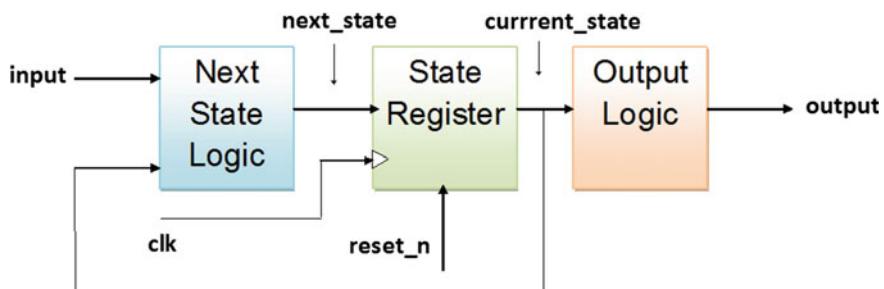


Fig. 4.5 Moore machine block diagram

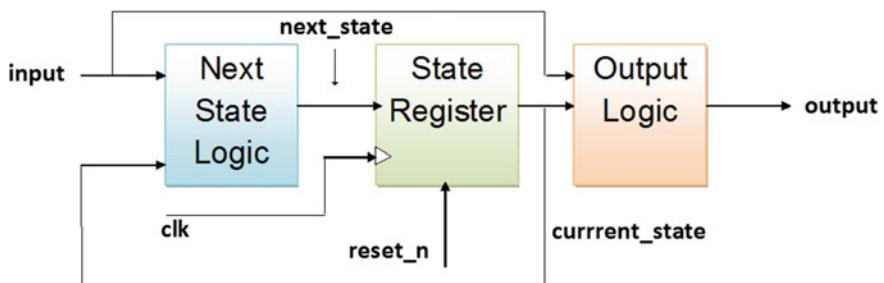


Fig. 4.6 Mealy machine block diagram

### 4.4.3 Moore Versus Mealy Machine

The differences between the Moore and Mealy machine is shown in Table 4.2.

#### How to improve the Performance of the FSMs?

To improve the performance of the FSMs, use the following techniques at the RTL level.

1. Do not use single ‘always’ block to code the FSMs as it does not yield into the efficient results.
2. To achieve the efficient synthesis for the FPGA/ASIC, use the multiple ‘always’ block. In practice, we can think of using
  - a. First always block for the next state logic
  - b. Second always block for the state register
  - c. Third always block for the output logic.
3. Use the blocking assignments inside the next state and output logic block as they are combinational in nature.
4. Use non-blocking assignments inside the state register block as this block is triggered on the active clock edge may be positive or negative.
5. Use the desired encoding method
  - a. Binary encoding needs  $n$  flip-flops for the  $2^n$  states
  - b. Gray encoding needs  $n$  flip-flops for the  $2^n$  states
  - c. One-hot encoding needs  $2^n$  flip-flops for the  $2^n$  states
6. To avoid the latch inference, use the default condition or cover all the case conditions in the case construct.
7. Depending on the number of transitions in the state machine use the if-else construct.
8. If area is not a bottleneck in the design for the clean timing uses one-hot encoding FSMs.

**Table 4.2** Moore Versus Mealy machine

Moore machine	Mealy machine
The output is function of current state only	The output is function of the current state and changes in the input
The output is stable for one clock cycle	Output may not be stable for one clock cycle as it is function of the input and current state
Output is not prone to glitches or spikes	Output is prone to glitches or spikes
STA is simple as the combinational paths are shorter. High operating frequency as compared to Mealy machine	STA is complex due to larger combinational path between registers. Less operating frequency as compared to Moore machine
More number of states as compared to Mealy machine	At least one state lesser as compared to Moore machine

9. For glitch-free output, use the registered output concept to register all the outputs for the clean timing.

## 4.5 RTL Design for Complex Designs

Imagine the complex design which consists of million or billion logic gates then the RTL design phase is time-consuming and iterative process. Prior to the RTL design, the design or product specifications need to be evolved based on the current need of the market and end customers.

For example, consider the mobile SOC design; then, the features required for mobile are high-speed processor, high-resolution display, keyboard, touchpad, internal memory, antenna, and external interfaces such as Bluetooth, USB, Wi-Fi functionality, memory controllers, power supply, clocking management, camera and mechanical assembly. By considering all these requirements, the design specifications are evolved prior to the RTL design phase.

The detail technical and functional specifications are captured in the system requirement and analysis document, and then the architecture of the design is created by the team of experts. During this phase, the design is partitioned into multiple blocks (can be called as initial floorplan of the chip). For example, analog blocks, digital blocks, memories, IP, video and audio processing functionality, and processor cores.

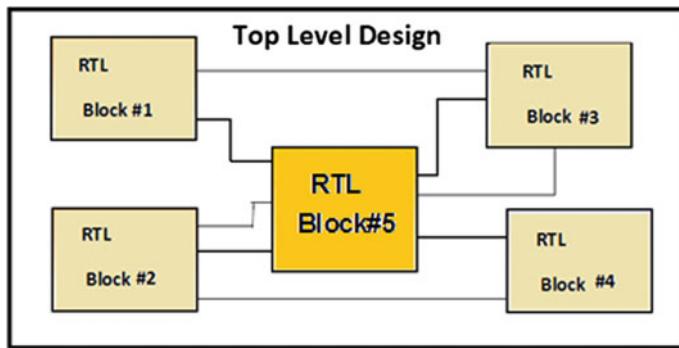
By considering all these; architecture evolves for the complex design and it is the block-level representation of the design. The micro-architecture of the design is always evolved from the architecture document, and during this phase the functional and timing details for each block are captured at the high level.

For the complex SOC designs depending on the requirements, the RTL design team can code the RTL for each functional block.

The functional verification can be carried out after the RTL design phase. But for the complex designs, the functional verification and RTL design phase can kick-start concurrently. Almost around 70% of the design cycle time and efforts are spent during the verification stage.

## 4.6 RTL Design at Top Level

The RTL design at the top level can be visualized as the integration of the number of functional blocks. As shown in Fig. 4.7 the design has the different functional blocks and the design is partitioned by keeping in mind the functionality of the design, interfaces, and hardware/software requirements. The RTL top module uses the instantiation of such functional blocks. For every RTL block, the area, speed, and power constraints need to be specified and finally at the top level the constraints are specified.



**Fig. 4.7** RTL at top level

The synthesis and design constraints are discussed in Chap. 9. The subsequent chapters discuss about the RTL design for the SOCs and the RTL verification.

## 4.7 Important Takeaways and Further Discussion

The following are the important points to summarize this chapter.

1. Use the architecture and micro-architecture document while coding using Verilog.
2. Use the RTL design guidelines.
3. Have understanding of the functionality of the design and their interfaces with the external blocks.
4. Use the tri-state, debug, and test logic at top level.
5. Use the synchronizers to pass the data between the multiple clock domains.
6. Use the divide and conquer approach for the large-density SOC blocks.
7. If IPs are used in the design, then understand the interfaces and have the wrappers for the connectivity.
8. Define the coverage goals for the constrained random verification.
9. Use the layered testbench and monitor the desired coverage.
10. Have the verification plan and test case generation at the block level and at the chip level.

The next chapter focuses on the architecture and micro-architecture evolution and design of the processor. The chapter is useful to understand how the large-density designs can be partitioned into the multiple functional blocks and how to think about the performance improvement strategies at the architecture level.

# Chapter 5

## Processor Cores and Architecture Design



*The architecture and micro-architecture document plays the important role during the design phase.*

**Abstract** The chapter describes the techniques to design the architecture and micro-architecture for the processor. The case study is created to develop the thought process to evolve the architecture of the pipelined processor. Most of the times, we need to have the processors in the SOC designs. For the complex designs, the processor IP cores can be used. The chapter's main objective is to develop the thought process of the engineers while sketching the architectures and micro-architectures for the processors. This can be helpful to design the products to implement and new ideas. The chapter is useful to understand the hard IP cores during SOC prototyping.

**Keywords** Processor · Data rate · Latency · Throughput · Pipelining · Bus · Ports Speed · Memory · SOC · Architecture · Micro-architecture  
Performance improvement

### 5.1 Processor Architectures and Basic Parameters

This section discusses the important parameters such as processor speed, clock rate, IO bandwidth, and multitasking.

#### 5.1.1 Processor and Processor Core

If we recall the history of the processor evolution, then we can notice that Intel launched a commercial processor 8080 during April 1974. It had non multiplexed

bus with separate address and data lines. Address bus was 16 bit wide, and data bus was 8 bit wide. The package used for this commercial processor was 40-pin DIP, and the clock frequency was 2 MHz.

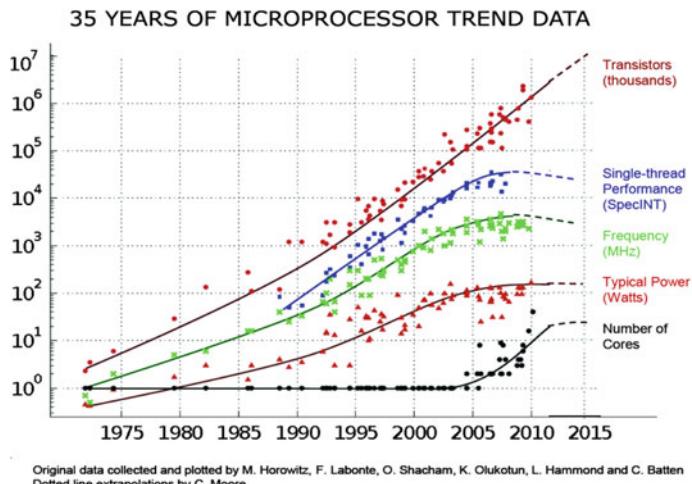
So if we consider the evolution of the processors, then the bus bandwidth, speed to achieve the concurrent execution plays an important role in the design of any SOC. During almost past four decades, the electronic system design using microprocessors has become the integral part of the embedded systems product development cycle to develop the innovative embedded systems. Use of the processor cores in the design reduces other components as thousands of transistors are placed on small silicon area to perform the required operations. With the evolution of the new algorithms and design techniques, the cost of the processor has reduced drastically and even the moderate functionality processors are available with less than 1\$ cost.

Figure 5.1 gives information about the 35 years of trend in the microprocessor design.

As shown in Fig. 5.1, the transistor count in thousands has increased exponentially from year 1975 to 2015 and even increasing further. Even the frequency of the processor during year 1975–1980 was less than 10 MHz, and during year 2015, the operating frequency of processor is almost few GHz.

Modern design during this decade is complex and dominated by the processor architecture. The companies like Intel, AMD, TI have the sophisticated processors with required functionality, internal memory, IO interfaces, and high-speed network interfaces. Even the processor has the flexible architectures to perform the complex floating point operations.

Effectively, the performance of the system is dependent on the performance of the processor. One of the techniques to improve the performance of the processor is by increasing the clock rate. If we recall year 1974, then the Intel 8080 speed was



**Fig. 5.1** Microprocessor trends

2 MHz, and during 1978, Intel 8086 speed was 10 MHz. So there was five times improvement in the speed during shorter span!

During year 1984, Intel launched 80386 processor which had speed of 25 MHz, and the Pentium II processor was launched by Intel during year 1994 which had speed of 266 MHz. As discussed, the speed of the modern processor is few GHz.

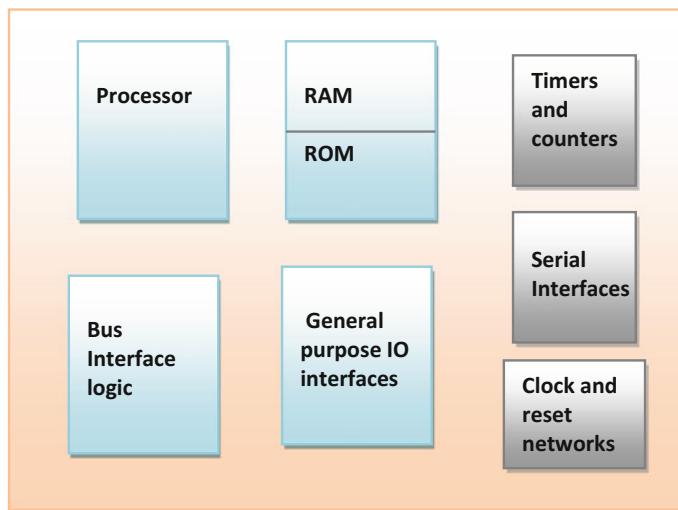
Another important factor in the design of the processor is data width and bandwidth of buses. As buses widened for each evolution, it has become easy to move the large amount of the data during each clock cycle. If we compare the Intel 16-bit processor 8086 with 80386, then the 8086 processor had 16 bit of data bus and 80386 had 32 bit of the data bus. Figure 5.2 shows the architecture of the SOC consisting of the

1. RAM
2. ROM
3. Processor
4. Serial IO
5. The general purpose functional logic

Using the SOC, the additional logic functionality can be interfaced using the programmable features.

During these days, the moderate gate count SOCs are available with little cost (may be around few \$). Another way to improve the performance of processor is by adding more buses as the shared bus performance is always lower as compared to the performance of multiple buses in the design.

If we consider the Intel Pentium II architecture, then it has separate cache memory bus. The processor can perform simultaneously two operations using main bus and cache memory bus.



**Fig. 5.2** Moderate gate count SOC

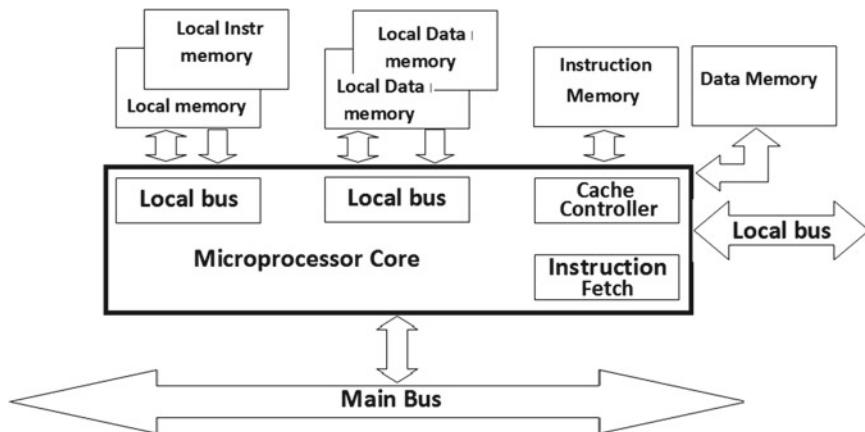
The impact of adding more number of buses in the processor architecture is on the pin count of the processor. So if we consider the evolution of processor, then in the past four decades the processor pin count has increased. Even as clock rate has increased over the period of time, it has impact on the power. So we can conclude that the processor power and energy density have increased exponentially in the past four decades.

As shown in Fig. 5.3, the performance of the processor can be increased by adding more number of buses. The performance enhancement technique such as use of high-bandwidth IO, buses, high clock rate are recommended in the design of SOC.

In the design of the processors, the additional pins means the higher cost of package and higher cost during testing. But in the SOC designs, the more number of pins costs nothing but they can incur the additional routing efforts. But once they are routed, then the additional pins does not add the significant cost of the chip. In the similar way, more number of buses cannot incur more cost but improves the design performance.

As shown in Fig. 5.3, the microprocessor core has more number of buses. The main bus communicates with microprocessor core and other buses communicate with the local instruction and data memories. There are additional buses to communicate with the instruction and data memory. The local bus shown is used to communicate with the high-bandwidth peripherals.

Due to use of these buses, the load/store of the data and instruction fetching can happen simultaneously. In addition to this, the cache controller can operate independently.



**Fig. 5.3** SOC architecture with multiple buses

### ***5.1.2 IO Bandwidth and Clock Rate***

If single processor is used in the design, then the IO bandwidth is fixed, but if processor core is designed for the SOC application, then it is possible to control the IO bandwidth. Due to use of multiple buses in the processor core, it is possible to improve the IO bandwidth and it is also possible to improve the data rate for high-speed data transfer. Concurrent IO data transfer can improve the design performance.

While architecting the SOC, the architect should think about the performance criteria in terms of area, speed, and power. If the architecture is designed using multiple buses operating at lower clock rate, then the power requirement will be less. But if the architecture uses the shared buses at higher clock rate, then the power requirement is more. So always there is trade-off between the area, speed, and power. It is always best practice to achieve the desired performance at lower clock rate. But if it is required, then it is essential to use the processor at higher clock rate.

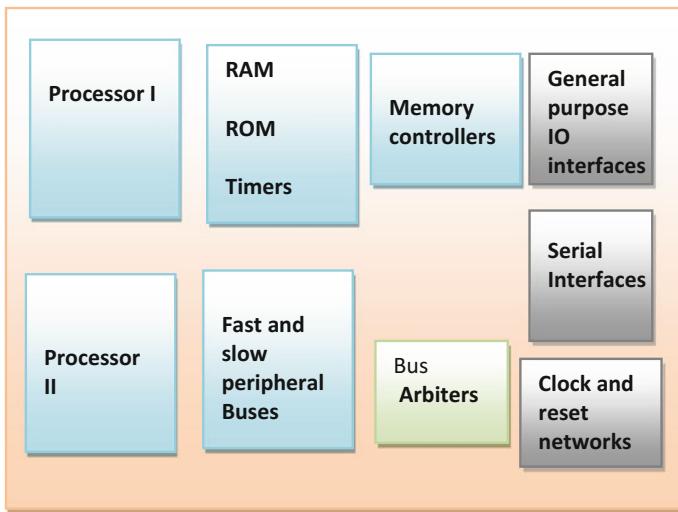
### ***5.1.3 Multitasking and Processor Clock Rate***

To improve the performance of the processor core, it is good practice to have the architecture with multitasking features. The basic technique to achieve the multitasking is by adding more queues in the design so that multitasking environment can be used to accomplish the task in specific time. Even multitasking can be achieved by increasing the clock rate of the processor. More the clock frequency indicates more number of operations performed concurrently.

So if we consider the single processor architecture where shared bus is communicating with the memories or IOs at higher clock rate, then the drawback is more power and additional overheads on the processor. As compared to this, it is better practice to use the fast and slow buses in the design (Fig. 5.4).

## **5.2 Processor Functionality and the Architecture Design**

Let us consider the 16-bit or 32-bit configurable processor core. The efficient architecture and RTL design for any SOC application is not only useful to improve the reliability of the processor, but also improves the overall performance in terms of area, speed, and power. The primary target in such kind of designs is to achieve the higher performance at the lower clock rates. As discussed earlier, we will use the Verilog to code the RTL. The processor design should have the feature of programmability, and it is better to have fewer instructions. The major risk in such type of designs is due to the changes in the specifications and functionality change to support the current market trends. Under such circumstances, the architecture should be designed to cope up these kinds of changes.



**Fig. 5.4** SOC with multitasking features

### Let us think what the processor core should have?

1. **Flexible architecture:** The architecture should be flexible enough to accommodate the changes during the design and implementation cycle. To have the better performance, if I wish to design the architecture, then, for the load or store instructions only the external memory interface can be used. For all other types of instructions, the inbuilt logic should be used which reduces the delays and improves the speed of the processor. External communication overheads should be as less as possible. This improves overall efficiency of the processor.
2. **Pipelining feature:** To improve the design throughput, the processor core should have the pipelined control logic. Depending on the requirements, the design should use the multistage pipelining. Special attention needs to be given to add the pipelined controlled stages in the design. If we consider the Intel processors and ARM processors, then the architectures have the multistage pipelining using the internal data and instruction queues.
3. **Internal storage:** The processor core architecture should have the enough internal storage. Most of the time, we encounter the architecture which has more number of general purpose registers. These general purpose registers can be used during execution of the load and store instructions. This type of the internal registers can hold the operand information required during the instruction execution.
4. **Simple instruction set:** For the improved throughput and performance of the core, the instructions should be single cycle and that can be accomplished by the pipelining. The better RTL implementation for the instructions used for the data transfer, arithmetic–logical operations and branching can result in the better performance.

5. **Operand definitions:** The design should have the operands for the source and destinations, and this kind of mechanism improves the control paths and their timing.
6. **External interfaces:** The processor core should be such that it should have the direct port interfaces, register interfaces, and interfaces for the FIFO to transfer the larger amount of data. Even the high-speed bus interfaces and network interfaces can be useful for the complex algorithms and data transfer.
7. **Port registers for the data transfer:** The performance of any processor is dependent on the IO bandwidth and the interfaces used in the transfer of the data. If any design needs multiple processor cores, then to transfer the data from one of the processor core to another, we can think of using direct port connections. Figure 5.5 shows the mechanism using the output and input registers.

The direct port connection to transfer the data from the processor #1 to processor #2 is shown in Fig. 5.5. As shown, when the #1 processor completes the instruction execution, then the result is stored in the output port register.

The contents of the output port register can be transferred to #2 processor and stored in the input register of #2 processor. To perform the data transfer initially the #1 processor executes the store instruction, and subsequently when the data need to be read by #2 processor, the #2 processor can initiate the load data transaction.

Even the queuing of the data is required while transferring data between the processors. As compared to the direct port interfaces, the high speed can be achieved.

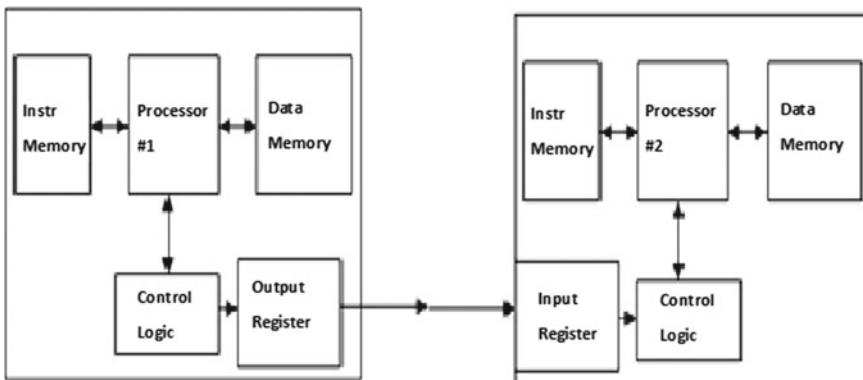


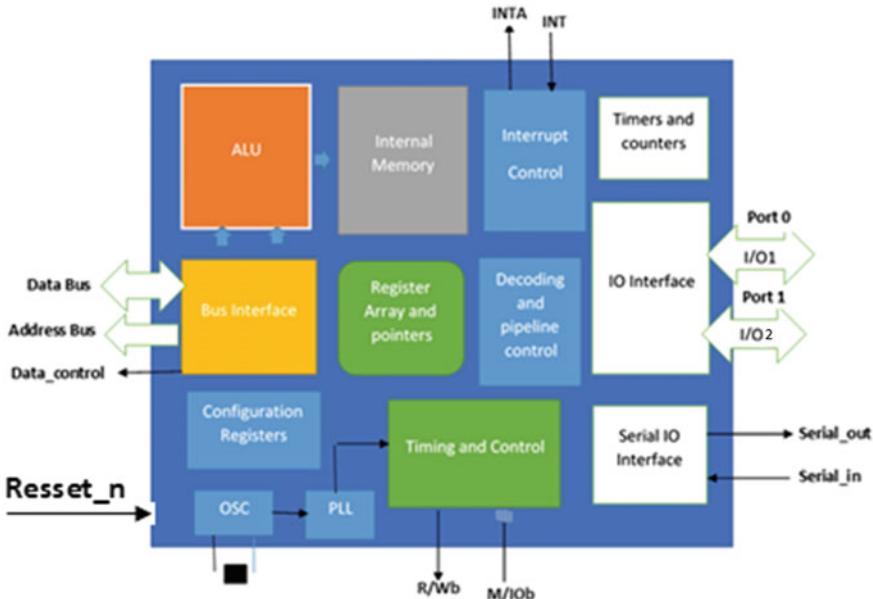
Fig. 5.5 Direct port connections

### 5.3 Processor Architecture and Micro-architecture

The architecture is the block-level representation for the design specifications. The architecture is developed from the functional specification of the design. Consider that for some application it is essential to design the architecture of the 16-bit processor. What we need to think?

1. **Application:** The environment in which processor works. Depending on that decide for the functionality. Even the market research to gather the processor functionality and improvement in them can play an important role.
2. **Operations:** What are the operation to be supported by the processor
  - a. Arithmetic
  - b. Logical
  - c. Data transfer
  - d. Branching
  - e. IO control.
3. **Size of data transfer:** Size of the data bus.
4. **Maximum addressable memory:** What should be the maximum addressable memory by the processor. Depending on that, extract the address bus count.
5. **Multiplexing of the address and data buses:** Provision for the multiplexed buses to reduce the processor pin count.
6. **Performance parameters:**
  - a. Speed
  - b. Power
  - c. Die size
  - d. Latency for the data transfer
  - e. Data rate
  - f. Throughput
  - g. Pipelined stages.
7. **Internal storage:** The storage required
  - a. Internal registers for temporary storage
  - b. Internal memory RAM, ROM, FIFO buffers.
8. **IO interfaces:**
  - a. General purpose IO ports
  - b. Serial interfaces
  - c. Network interfaces
  - d. High-speed bus interfaces.

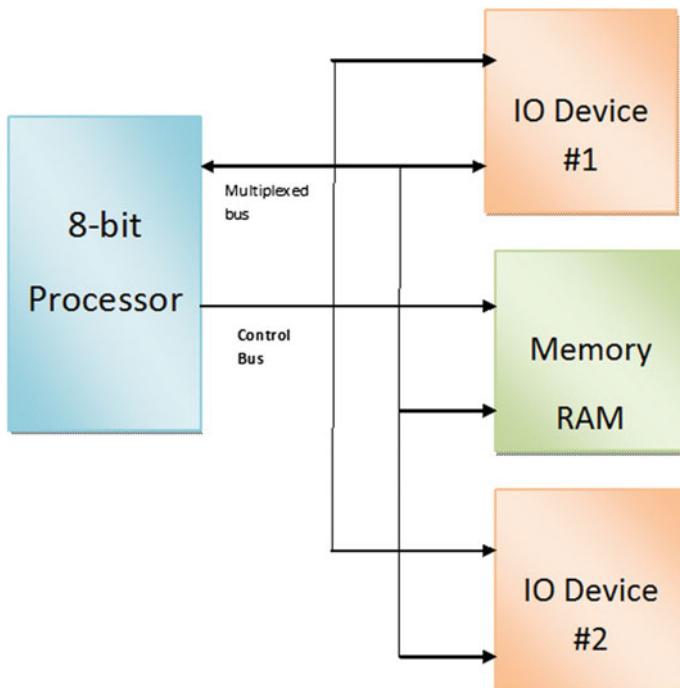
By using all these parameters, the block-level functionality can be documented and shown in Fig. 5.6. It is the basic architecture which is the initial phase to kick-start the design. In the practical SOC implementations, the architecture design team needs to have the good amount of experience and the imagination. The team of experts can create the architectures for the pipelined processor, and it is iterative process.



**Fig. 5.6** Processor architecture at basic level

The architecture document should consist of the information about the

1. **Functionality of each block**
  - a. ALU: 16-bit ALU to perform the arithmetic–logical operations
    - i. The instruction type: arithmetic, logical, data transfer.
2. **The information about the internal storage**
  - a. Register array
  - b. Internal memory.
3. **The high-level information about the data flow**
  - a. Fetch the instruction: Bus interface logic is used.
  - b. Decode the instruction: Decoding logic is used.
  - c. Execute the instruction: ALU and other associated logic depending on the type of instruction.
  - d. Store the result.
    - i. In the internal register/memory
    - ii. External memory.
4. **Information about the initialization/configuration and test registers and logic:** Test and debug logic for the initialization and configuration.
5. **High-level information about the external interfaces (serial and parallel):**



**Fig. 5.7** Multiplexed buses

- a. Serial interfaces
- b. General purpose IO interface.
6. **Interrupts:** Execution of immediate tasks
  - a. Number of interrupts (if more than one).
7. Information about the clock reset network.
8. Information about the constraints.
9. External world connectivity (for the external high-speed interfaces if any).
10. Information about the power supplies and voltage domains.

#### Pin count:

To get the more external pin visibility, the architecture document can have the details of the width of each pin.

1. For example, the external memory interfaced to the processor is 1 MB; then, the width of address bus should be 20 bit.
2. The size of the data bus is 16 bit. Size of each register is 16 bit. Size of the memory pointers is 20 bit.

**The pin count minimization:** Use the multiplexed address data bus which is as shown in Fig. 5.7.

**Table 5.1** External world connectivity

External_interface	Pin_count	Description
Data_bus	16	Bidirectional bus to transfer the data
Address_bus	20	For the IO or memory address
Data_control	1	For demultiplexing of the address data lines
R/Wb	1	For read and write. For read status on this line is logic ‘1’ and for write status on this line is logic ‘0’
M/IOb	1	The output pin and logic ‘1’ status indicates the memory operation, and logic ‘0’ on this line indicates IO operation
Crystal_input	2	Crystal input pins
Serial_in	1	To connect serial-input device
Serial_out	1	To connect serial output device
INT	1	Level-sensitive interrupt to the processor
INTA	1	Interrupt acknowledge from the processor
Reset_n	1	Active low reset input

Table 5.1 gives information about the block interfaces with the external devices.

In addition to this, the processor should have the power supply connections. For this discussion, it is ruled out.

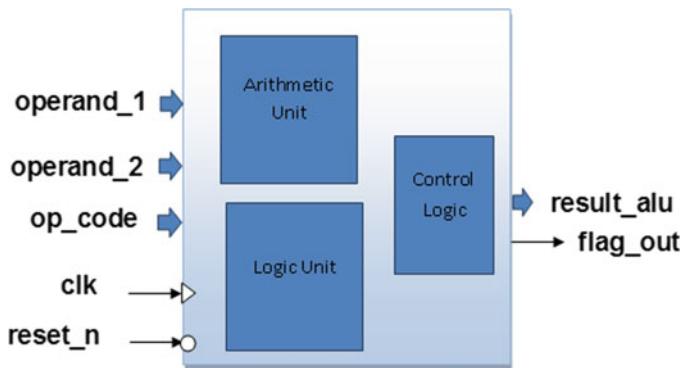
### 5.3.1 Processor Micro-architecture

The micro-architecture for the processor is the sub-block-level representation of the individual functional blocks. The micro-architecture should give information about the high-level logic requirement for the individual functional blocks and their timing and interface details. The following section elaborates this in much more detail. For the high-density functional blocks, it may be difficult task, but the micro-architecture evolution is helpful during the RTL design, verification, and implementation phases.

#### 5.3.1.1 ALU

It performs the arithmetic and logical operations and operates on the two operands. What should we think while developing the micro-architecture for the ALU?

1. Size of the operand.
2. Is it allow the pipelined execution of the instructions? If not, the single-cycle execution is possible for which type of instructions.
3. How many logical instructions and arithmetic instructions it supports?



**Fig. 5.8** ALU micro-architecture

4. What kind of the status/initialization information is required to perform the operations?
5. Whether it generates information about the status as overflow, zero result, etc.?

By using this thought process, the micro-architecture for this block can be evolved.

As shown in Fig. 5.8, the ALU block is partitioned into the arithmetic unit and logical unit. The control logic takes decision to perform either arithmetic or logical instructions depending on the status of the opcode decoding logic.

During the RTL design phase, this can play important role. The RTL engineer can think about using the multiple blocks to partition the design. Whether the design size is moderate or complex, this can give the better clarity about the use of the HDL constructs to execute only one instruction at a time.

1. **Interblock dependability:** Have a thought process about the interblock dependability and high-level timing. Consider the following:
  - a. The instruction decoding should generate the `op_code` for the instruction.
  - b. The `operand_1` and `operand_2` need to be fetched.

Table 5.2 gives information about the signals for this block.

**Risk:** The single-cycle execution is possible for the addition, subtraction, logical instructions. The risk is the multiplication and division algorithms. It requires the extra feature and pipelined support.

**Table 5.2** ALU signal connectivity at top level

signal_name	Width	Description	Direction
clk	1	Common clock signal	Input
reset_n	1	Active low asynchronous reset	Input
operand_1	16	Operand 1	Input
operand_2	16	Operand 2	Input
op_code	4	4 bit for the 16 instructions	Input
result_alu	16	16-bit output from ALU	Output
Flag_out	1	To indicate the overflow	Output

### 5.3.1.2 Serial IO Interface

The serial devices can communicate with the processor using serial input and output line. To get the sub-block-level understanding, we can think of:

1. Maximum serial IO data rate.
2. Maximum clock frequency of the serial IO versus processor clock speed.
3. How to segregate the data from serial input to get the required parallel data.
4. How to transfer the parallel data into serial form.

So at the high level, the logic can be developed using the bidirectional shift registers.

As shown in Fig. 5.9, the bidirectional shift register is used to communicate with the serial IO; the direction control logic depending on the IO instruction decides about the data transfer to or from the processor.

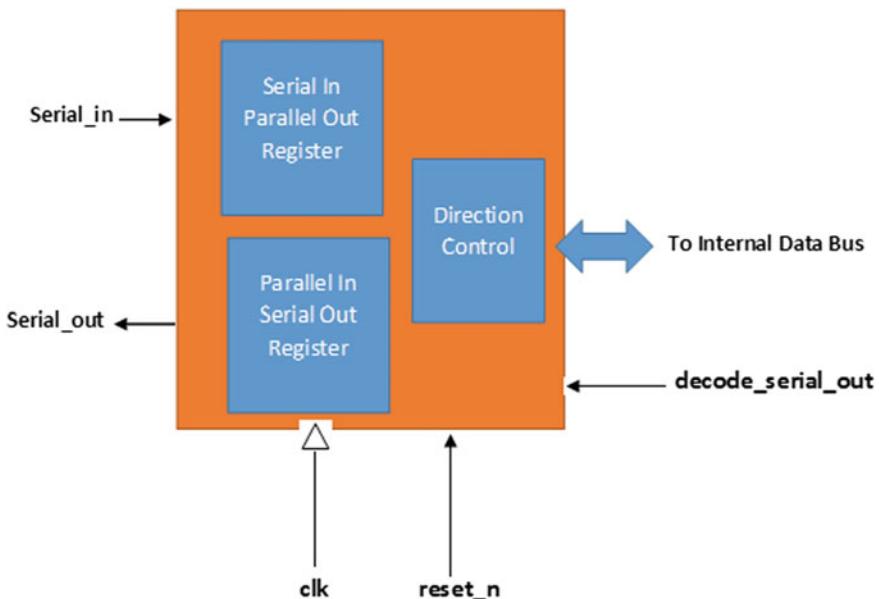
During the RTL design phase, this is helpful to code the design using two different procedure blocks. One can be used to sample the input data, and other can be used to transfer the serial data. The care should be taken while writing the RTL for the direction control. This type of logic uses the sequential shift registers to sample or transfer the data.

The serial interface signal information is given in Table 5.3.

### 5.3.1.3 Internal Registers

The registers can be used to store the result or can act as operands. The micro-architecture can be viewed as parallel-in parallel-out (PIPO) registers to have the read and write control. Depending on the instruction type, the data can be transferred or stored in these registers. Even the special purpose pointers can be of PIPO type.

What is the additional logic with these registers?



**Fig. 5.9** Serial IO micro-architecture

**Table 5.3** Serial interface connectivity

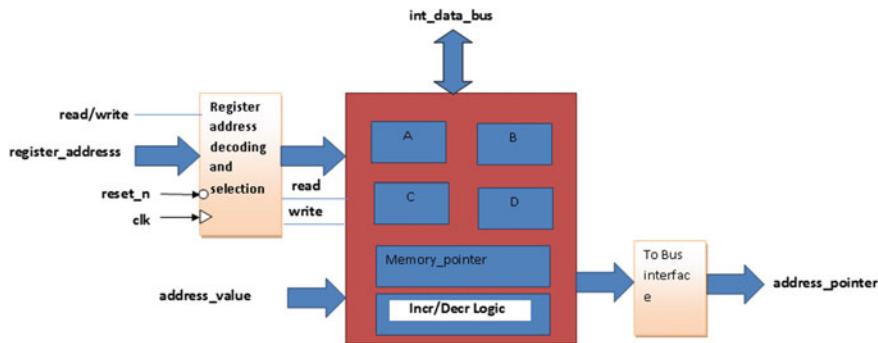
signal_name	Width	Description	Direction
clk	1	Common clock signal	Input
reset_n	1	Active low asynchronous reset	Input
Serial_in	1	Serial input to the logic	Input
Serial_out	1	Serial output to the logic	Output
Data_inout	16	Bidirectional data bus to transfer the data	Bidirectional
decode_serial_out	1	Input to the block from decoding logic	Input

1. Register selection logic that is decoding logic.
2. The direction control using the signals from the control and timing unit.

As shown in Fig. 5.10, the sub-block representation has the decoding logic, memory, and internal memory pointers.

During the RTL design phase, this is helpful to code the design. Write the RTL using different procedure blocks for the following:

1. PIPO logic for the registers and pointers
2. Direction control
3. Address decoding logic
4. Read/write and address pointer logic



**Fig. 5.10** Internal registers and pointers

The table gives information of the interfaces for this functional block (Table 5.4).

#### 5.3.1.4 Interrupt Control

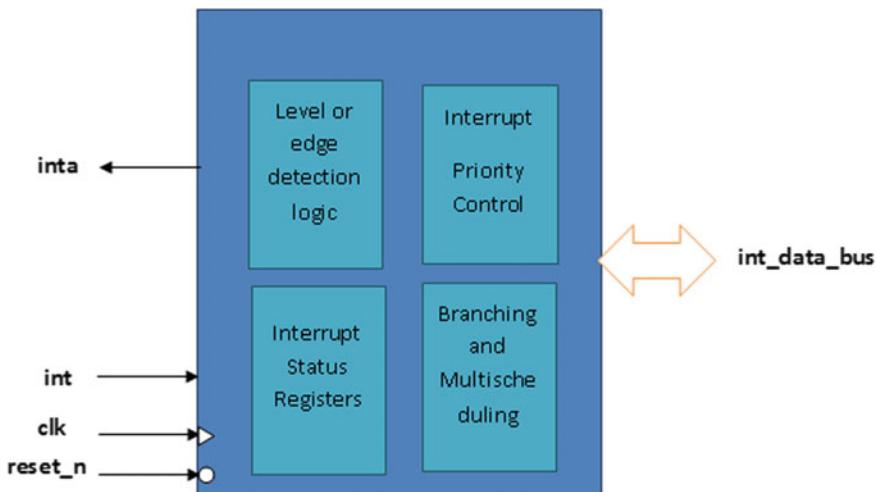
To develop the micro-architecture for such kind of the logic, the thought process can be

1. The type of interrupts (edge triggered or level-sensitive)
2. What is the priority of the interrupts (in case of multiple hardware interrupts)
3. Enabling and disabling of interrupts and the logic
4. The vector location (branching address) logic (if not supported by any other mechanism)

As shown in Fig. 5.11, the sub-block-level representation can sense the level or edge, detects the priority, and processes the interrupt depending on the status of the interrupt enable (Table 5.5).

**Table 5.4** Interrupt control interface

signal_name	Width	Description	Direction
clk	1	Common clock signal	Input
reset_n	1	Asynchronous active low input	Input
register_address	2	To select one of the register at a time	Input
read/write	1	Input to indicate read or write	Input
int_data_bus	16	Read or write data from/to the register	Bidirectional
address_value	20	Input to the memory pointer	Input
address_pointer	20	The external memory bus address	Output



**Fig. 5.11** Interrupt control logic

**Table 5.5** Internal registers and pointer interfaces

signal_name	Width	Description	Direction
clk	1	Common clock signal	Input
reset_n	1	Active low asynchronous input	Input
register_address	2	To select one of the register at a time	Input
int	1	Input to indicate the level-sensitive input interrupt	Input
int_data_bus	16	Read or write data from/to the status register	Bidirectional
inta	1	Interrupt acknowledge	Output
address_pointer	20	The external memory bus address	Output

During the RTL design phase, this is helpful to code the design. Write the RTL using separate procedure blocks

1. Edge or level detection logic
2. Priority detection logic
3. Interrupt status and branching/scheduling of the interrupt logic.

### 5.3.1.5 Decoding and Control and Timing

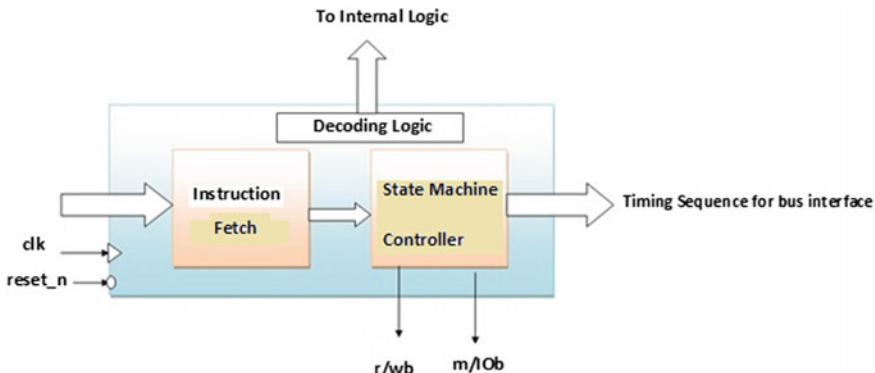
To develop the micro-architecture for such kind of the logic, the thought process can be

1. What is the type of instruction and op-code decode?
2. What types of the internal signals need to be derived?
3. What type of the external control and timing signals need to be derived?
4. What should be the timing of the signals?

As shown in Fig. 5.12, the sub-block-level representation is shown and can fetch the instruction, decode the instruction and the state machine controller can generate the control and timing signals (Table 5.6).

During the RTL design phase, this is helpful to code the design. Write the RTL using different procedure blocks

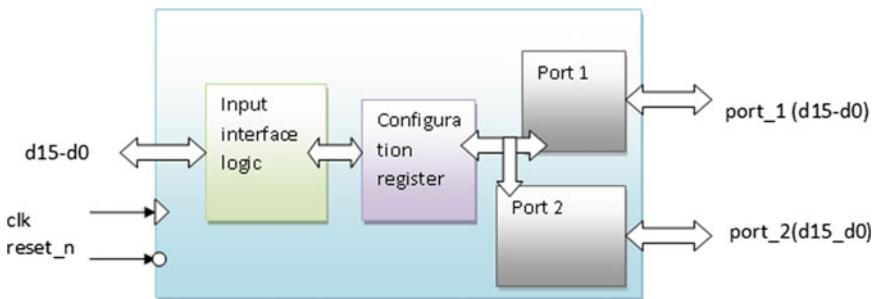
1. Fetch the opcode.
2. Create the state machine controller.
3. Implement the decoding logic.



**Fig. 5.12** Decoding and control and timing signals

**Table 5.6** Decoding and timing control logic

signal_name	Width	Description	Direction
clk	1	Common clock signal	Input
reset_n	1	Active low asynchronous reset	Input
Instruction_code	4	The 4-bit operational code of the instruction	Input
rd/wb	1	The output line to indicate the read transaction for logic ‘1’ and write transaction for logic ‘1’	Output
M/IOb	1	Logic ‘1’ on this line indicates operation with memory and logic ‘0’ indicates IO operation	Output



**Fig. 5.13** IO interface logic

**Table 5.7** IO interface

signal_name	width	Description	Direction
clk	1	Common clock signal	Input
reset_n	1		Input
D15-D0	16	The 16-bit bidirectional bus	Bidirectional
Port_1(D15-D0)	16	Bidirectional IO port	Bidirectional
Port_2(D15-D0)	16	Bidirectional IO port	Bidirectional

### 5.3.1.6 IO Interfaces

To develop the micro-architecture for such kind of the logic, the thought process can be

1. The type of IO operation
2. Port register selection logic
3. Direction control and configuration registers

The logic is shown in Fig. 5.13 and the description of associated signals is given in Table 5.7).

During the RTL design phase, this is helpful to code the design. Write the RTL using different procedure blocks

1. Fetch the configuration and status information.
2. Implement the direction control using the configuration logic.
3. Design the port registers for the bidirectional communication.

### If my design is complex or if I have product idea, then what to do?

For the complex billion gate design, the micro-architecture sketching is not feasible according to the process explained in the above section. The team can think about the following:

1. Team of architect brainstorms and think about the functional blocks.
2. For complex design, find the hard or soft core IPs required. They can be open source or licensed version.
3. Create the initial top-level floor plan for the design.
4. For each functional block
  - a. For IP used
    - i. Understand the functionality and timing information.
    - ii. Understand the interfaces.
    - iii. Understand about the configuration.
  - b. If the architecture is available
    - i. Check for the required modifications.
    - ii. Check for the interfaces and wrappers required.
  - c. If architecture of the block is not available
    - i. Use hardware and software partitioning.
    - ii. Sketch the block level representation.
    - iii. Sketch the micro-architecture.
  - d. Estimate the external IO interfaces and general purpose interfaces.
  - e. Have understanding of the latency, data rate, and throughput.
  - f. Clock and reset logic required.
  - g. Multiple voltage domain and power domains in the design.

## 5.4 RTL Design and Synthesis Strategies

The design of the processor can be efficiently achieved using the modular design approach or the bottom-up approach. Have the design partitioning at the architecture level, use the reference document as micro-architecture, and code the RTL for the

1. Top\_RTL.v
  - a. ALU.v
  - b. Internal\_memory.v
  - c. Bus\_interface.v
  - d. Internal\_registers.v
  - e. Interrupt\_control.v
  - f. Timing\_control.v
    - i. Timers\_counter.v
  - g. Serial\_IO\_control.v

Use the synchronizers, tri-state logic, and clock reset network in the top block.

### 5.4.1 *Block-Level Design*

The team members (RTL design, verification, synthesis, and STA team) can perform the following

1. Code the efficient RTL for the each functional block.
2. Verify the design using the required tests.
3. Use the block-level constraints and synthesize the design.
4. Check whether the constraints can meet or not?
5. Perform the timing simulation and prelayout STA. Fix the setup time violations using the RTL tweaks and architecture tweaks.

### 5.4.2 *Top-Level Design*

The team can perform the following tasks:

1. Create the Top.v using the instantiation and verify the design.
2. Check whether the coverage goals met or not?
3. Perform the design synthesis using top-level constraints.
4. Check whether the constraints met or not?
5. Perform the prelayout timing for the top.v.
6. Check for the timing violations and fix them using the necessary tweaks.

## 5.5 Design Scenarios

The objective of this section is to discuss the frequently encountered design scenarios during the RTL design stage of the processor. Depending on the functional specifications and requirements, the designer can modify the RTL to realize the functionality.

### 5.5.1 *Scenario 1: Instruction Set and ALU Design*

ALU instructions and execution: The processor instruction set and the RTL coding. Consider that the processor has the following arithmetic and logical instructions:

1. Transfer (a\_in)
2. Addition without carry (a\_in, b\_in)
3. Addition with carry input (a\_in, b\_in, cin)
4. Subtract without borrow (a\_in, b\_in)
5. Subtract with borrow (a\_in, b\_in, cin)
6. Increment by 1 (a\_in, 1)

7. Decrement by 1 (a\_in, 1)
8. OR (a\_in, b\_in)
9. XOR (a\_in, b\_in)
10. AND (a\_in, b\_in)
11. NOT (a\_in)

Let us consider the RTL design for these instructions. If more instructions needs to be supported, then use the multiple modules, that is, modular design approach

```
module alu_design (clk , reset_n ,op_code , a_in ,b_in,cin,y_out, cout);
input clk;
input reset_n;
input [3:0] op_code;
input [15:0] a_in,b_in;
input cin;
output reg cout;
output [15:0] y_out;
reg [15:0] y_out;

always @ (posedge clk or negedge reset_n)
begin

if (~reset_n)
{cout,y_out} = 0;

else

case (op_code)

4'b0000 : {cout, y_out }= {0,a_in};
4'b0001 : {cout, y_out }= a_in+b_in;
4'b0010 : {cout, y_out }= a_in+b_in+cin;
4'b0011 : {cout, y_out }= a_in -b_in;
4'b0100 : {cout, y_out }= a_in -b_in-cin;
4'b0101 : {cout, y_out }= a_in +1'b1; ----->
4'b0110 : {cout, y_out }= a_in -1'b1;
4'b1000 : {cout, y_out }= {0, (a_in | b_in)};
4'b1001: {cout, y_out }= {0,(a_in ^ b_in)};
4'b1010: {cout, y_out }= {0,(a_in & b_in)};
4'b1011: {cout, y_out }= {0,~a_in} ;
default : {cout,y_out} = 0;

endcase

end

endmodule
```

- Use of the registered input and registered output for the alu functionality
- Used case construct so that only one instruction is executed at a time.
- Enable the resource sharing options in the EDA tools.

**Example 5.1** Synthesizable Verilog code for ALU using case

```

module alu_logic ( clk, reset_n,op_code, a_in,b_in,cin, y_out, cout);
  input clk;
  input reset_n;
  input [3:0] op_code;
  input [15:0] a_in,b_in;
  input cin;
  output reg cout;
  output [15:0] y_out;
  reg [15:0] y_out;

  always @(posedge clk or negedge reset_n)

  begin

    if (~reset_n)
      {cout,y_out} = 0;

    else
      if (op_code==4'b0000)
        {cout, y_out }= {0,a_in};

      else if (op_code==4'b0001)
        {cout, y_out }= a_in+b_in;

      else if (op_code==4'b0010)
        {cout, y_out }= a_in+b_in+cin;

      else if (op_code==4'b0011)
        {cout, y_out }= a_in - b_in;

      else if (op_code==4'b0100)
        {cout, y_out }= a_in - b_in - cin;

      else if (op_code==4'b0101)
        {cout, y_out }= a_in + 1'b1;

      else if (op_code==4'b0110)
        {cout, y_out }= a_in - 1'b1;
      else if (op_code==4'b1000)
        {cout, y_out }= a_in|b_in;

      else if (op_code==4'b1001)
        {cout, y_out }= a_in ^ b_in;

      else if (op_code==4'b1010)
        {cout, y_out }= a_in & b_in;
  end

```

- Use of the registered input and registered output for the alu functionality
- Used if-else construct in the RTL.
- Enable the resource sharing options in the EDA tools.

**Example 5.2** Synthesizable Verilog code using nested if-else

```

else if (op_code==4'b1011)
{cout, y_out }=~ a_in;

else
{cout, y_out }=16'b0;

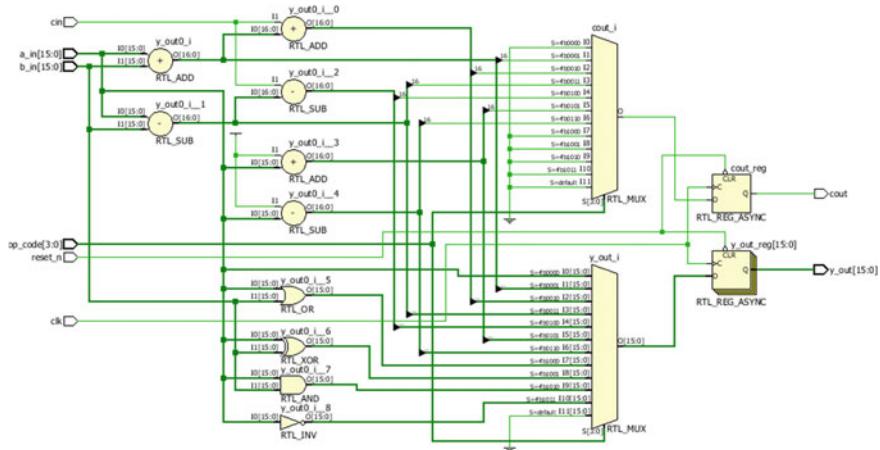
end

endmodule

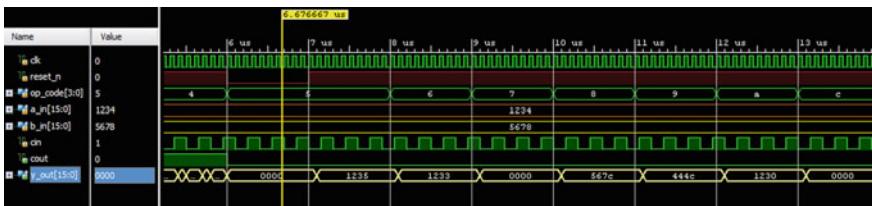
```

➤ Not recommended to use the if-else construct for such kind of designs as it infers priority logic.

**Example 5.2** (continued)



**Fig. 5.14** Synthesis result for the ALU using case construct



**Fig. 5.15** Simulation result of 16-bit ALU

for the better timing and synthesis results. Use the registered inputs and registered outputs (Examples 5.1 and 5.2, Figs. 5.14 and 5.15).

### 5.5.2 Scenario 2: Data Load and Shifting

They are used to load parallel data and to perform the right or left shift operation (Table 5.8).

The Verilog code is described in Example 5.3 (Fig. 5.16).

**Table 5.8** Shift operations

Select code	Operation
00	Load parallel data
01	Right shift by 1 bit
10	Left shift by 1 bit
11	Hold the data

```
module shift_register ( clk, reset_n, op_code, data_out, data_in, MSB_out, LSB_out);

input clk;
input reset_n;
input [1:0] op_code;
input [15:0] data_in;
output MSB_out, LSB_out;
output [15:0] data_out;
wire [15:0] data_out;
reg [15:0] tmp_data_out;

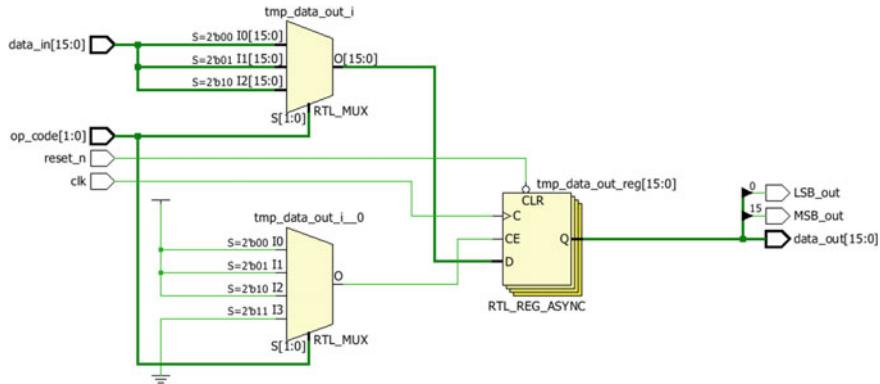
always @((posedge clk or negedge reset_n))

begin
if (~reset_n)
tmp_data_out <= 16'b0;
else
  case (op_code)
    2'b00 : tmp_data_out <= data_in;
    2'b01 : tmp_data_out <= {data_in[0], data_in[15:1]};
    2'b10 : tmp_data_out <= {data_in[14:0], data_in[15]};
    2'b11 : tmp_data_out <= tmp_data_out;
  endcase
end

assign data_out = tmp_data_out;
assign MSB_out = tmp_data_out[15];
assign LSB_out = tmp_data_out[0];
endmodule
```

- Depending on the operational code status the output of shift register is generated.
- The parallel output is designated as data\_out
- Serial output for the left shift operation is taken from MSB\_out
- Serial output for the right shift operation is taken from the LSB\_out

**Example 5.3** Synthesizable Verilog code for shift register



**Fig. 5.16** Synthesis result of shift\_register

### 5.5.3 Scenario 3: Parallel Data Load

The parallel-input and parallel-output register is used to fetch the parallel data and generate the parallel data output depending on the status of enable input. Use the similar kind of strategy for the instruction register and address register. The Verilog code is described in Example 5.4 (Fig. 5.17).

### 5.5.4 Scenario 4: Serial Data Processing

The serial-input serial-output register is used to establish serial data communication. The Verilog code is described in Example 5.5 (Fig. 5.18).

### 5.5.5 Scenario 5: Program Counter

The program counter used to point the next instruction while executing the present instruction. The program counter increment logic is described using Example 5.6 (Fig. 5.19).

### 5.5.6 Scenario 6: Register Files

The register files can be used to store the data. The Verilog code is described in Example 5.7 (Fig. 5.20).

```

module paralle_in_parallel_out ( clk, reset_n, enable_in, data_in, data_out);

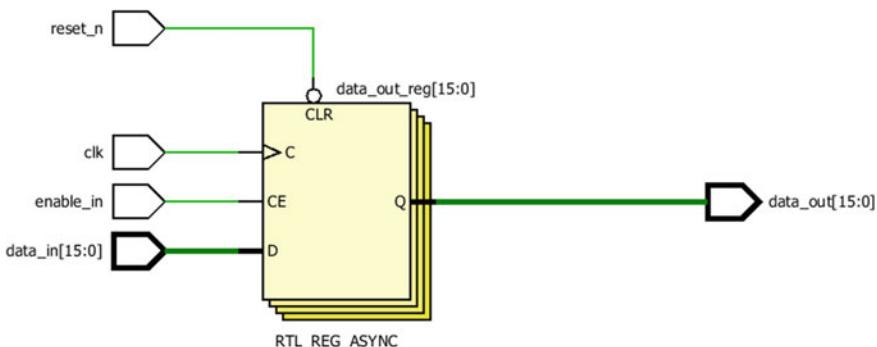
input clk;
input reset_n;
input enable_in;
input [15:0] data_in;
output [15:0] data_out;
reg [15:0] data_out;

always @(posedge clk or negedge reset_n)
begin
if (~reset_n)
data_out<= 16'b0;
else if (enable_in)
data_out<= data_in;
end
endmodule

```

- For enable\_in='1' the parallel data\_in is loaded in the register.
- If enable\_in='0' it holds the previous data.
- The logic generates the 16 bit register sensitive to rising edge of clock with multiplexer logic to sample the data.

**Example 5.4** Synthesizable Verilog code for the parallel data processing



**Fig. 5.17** Synthesis result for parallel-input parallel-output register

```

module serial_in_serial_out ( clk, reset_n, enable_in, data_in, data_out);

input clk;
input reset_n;
input enable_in;
input [15:0] data_in;
output data_out;
wire data_out;

reg [15:0] tmp_data_out;

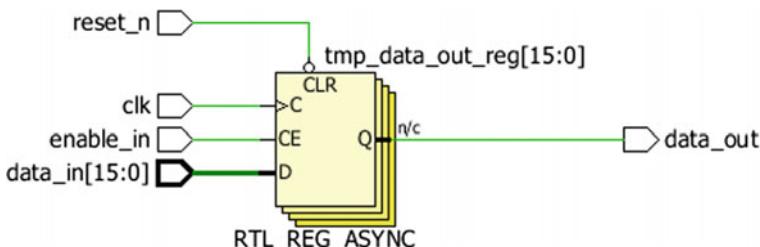
always @(posedge clk or negedge reset_n) begin
    if (~reset_n)
        tmp_data_out<= 16'b0;
    else if (enable_in)
        tmp_data_out<={data_in[0], data_in[15:1]};
end

assign data_out = tmp_data_out[0];

```

- For enable\_in='1' the parallel data\_in is loaded in the register.
- If enable\_in='0' it holds the previous data.
- The logic generates the 16 bit register sensitive to rising edge of clock with multiplexer logic to sample the data.

**Example 5.5** Synthesizable Verilog code for the serial data processing



**Fig. 5.18** Synthesis result of serial-input serial-output shift register

## 5.6 Performance Improvement

The processor performance can be improved at the architecture level by adding the pipelined stages and by improving the clock rate and IO bandwidth as discussed.

```

module program_counter ( clk, reset_n, pc_in,load_pc, incr_pc, pc);
parameter size=16;
input clk;
input reset_n;
input load_pc;
input incr_pc;
input [size-1:0] pc_in;
output [size-1:0] pc;
reg [size-1:0] pc_out;
always @(posedge clk or negedge reset_n)
begin
if (~reset_n)
pc_out<= 16'b0;
else if (load_pc)
pc_out<=pc_in;
else if (incr_pc)
pc_out<=pc_out+1;
end
assign pc=pc_out;
endmodule

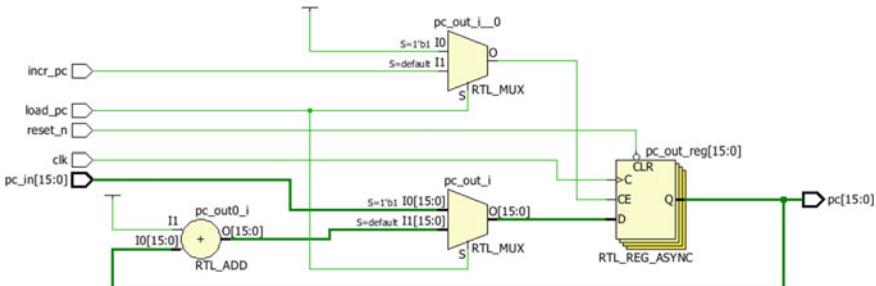
```

- For load\_pc='1' the pc\_in is loaded in the program counter.
- For the incr\_pc ='1' the program counter is incremented by 1
- The logic generates the 16 bit register sensitive to rising edge of clock with multiplexer logic for the increment and the load.

**Example 5.6** Program counter synthesizable Verilog code

Consider the following four instructions which need to be executed in sequence:

Add reg0, reg1, reg7	$(\text{reg}0) + (\text{reg}1) = (\text{reg}7)$
Sub reg2, reg3, reg6	$(\text{reg}2) - (\text{reg}3) = (\text{reg}6)$
Load 16 bit data, reg5	16bit data = (reg5)
Store reg4, Memory_Loc	$(\text{reg}4) = (\text{Memory\_Loc})$

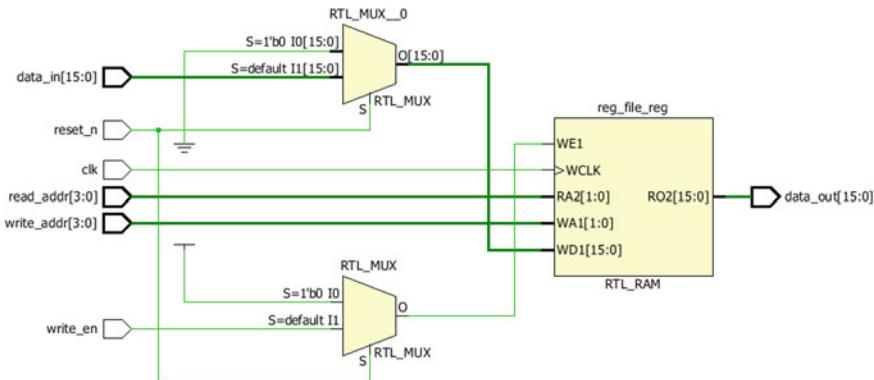


**Fig. 5.19** Synthesis result for program counter

```
module register_file ( clk, reset_n, write_addr, write_en, data_in, read_addr, data_out);
parameter size=16;
parameter addr=4;
input clk;
input reset_n;
input [addr+1:0] write_addr, read_addr;
input write_en;
input [size-1:0] data_in;
output [size-1:0] data_out;
reg [size-1:0] reg_file [0 : addr];
always @((posedge clk or negedge reset_n)
begin
if (~reset_n)
  reg_file [write_addr] <= 16'b0;
else if (write_en)
  reg_file[write_addr] <= data_in;
//pc_out<=pc_out+1;
end
assign data_out=reg_file[read_addr];
endmodule
```

- For `write_en='1'` the parallel `data_in` is loaded in the register file.
- Depending on the status of the `read_addr` the data stored in the register file is outputted on the `data_out`.

**Example 5.7** Register file synthesizable Verilog



**Fig. 5.20** Synthesis result of register file

Consider that each instruction needs to go through fetch, decode, execution, and store result; then, without pipelining, it will take 4 clock cycles. That means for the four instructions, it will take 16 clock cycles. Due to use of the pipelined control logic, if four-stage pipelined is incorporated in the design, then it will reduce the number of clock cycles and improve the performance of the design.

The Table 5.9 illustrates the execution of the four instructions.

As shown in Table 5.9 to store the result of the Add instruction, it takes four clock cycles. But due to pipelining, the second instruction onwards it will utilize less number of clock cycles. For the four instructions, the result is available in the 7 clock cycles and that is a performance improvement by 8 clock cycles.

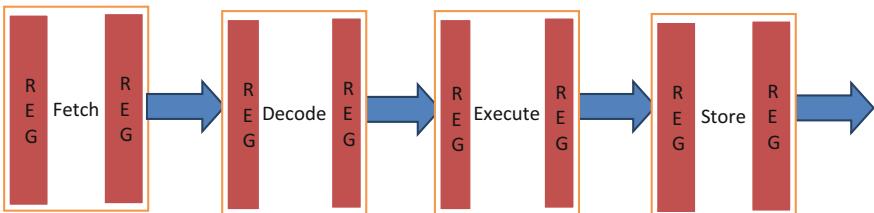
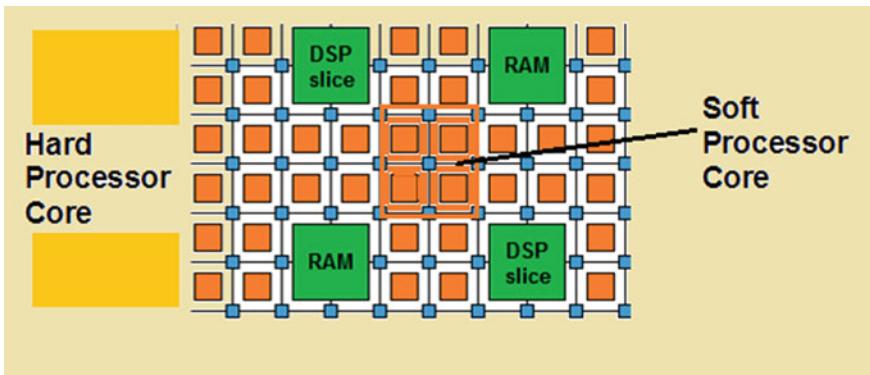
The pipelined stage using the registered outputs and inputs is as shown in Fig. 5.21.

### 5.6.1 How to Tweak the RTL to Improve the Design Performance

Due to pipelining, the design performance can be improved. Commonly used techniques to improve the design performance using the pipelining concept are register balancing and register optimization. Depending on the requirement of the hierarchical designs or flattened design, these techniques can be used during the RTL design and synthesis phase. The RTL can be tweaked by adding the pipelined stages; this increases the latency for the design by improving the register-to-register path timing.

**Table 5.9** Instruction pipelining

Clock cycle	Fetch	Decode	Execute	Store result
I	Add	X	X	X
II	Sub	Add	X	X
III	Load	Sub	Add	X
IV	Store	Load	Sub	Add

**Fig. 5.21** Pipelined stage**Fig. 5.22** Hard and soft processor cores on FPGA fabric

## 5.7 Use of Processors in SOC Prototyping

Most of the modern FPGAs use the hard processor cores. If you think about use of the Xilinx or Intel FPGAs, then the ARM-based hard processor core architectures are inbuilt in the FPGA. They can be used during the prototyping. For the details of the Xilinx and Intel FPGA architectures, refer Chaps. 11, 12 and 15.

The soft and hard processor cores used in most of the complex designs work at the speed of the 100 MHz and above. The prototype team can use the soft processor cores running at high speed of 200–250 MHz if required. The hard processor cores which are available on the FPGA fabric can run at the speed of the 100 MHz for most of the available high-density FPGAs from Xilinx and Intel. The functionality and timing of such cores decide the performance of the prototyping (Fig. 5.22).

**Table 5.10** Soft versus hard processor cores

Features	Soft core	Hard core
Flexible architecture	Tweaking and addition of the IP is very much possible. The addition of the external interface components is possible with the soft IP cores	It is hard core, and the architecture is fixed. The addition of the external components/IPs like ADC, DAC is not possible. Using the add-on boards or the design partitioning and interfacing, this can be accomplished
Operating frequency	High (around 250–300 MHz)	Moderate (around 100–150 MHz)
Logic density	Moderate	High density
The visibility to internal logic	Access of the internal signals using the logic analyzer or oscilloscope during prototype is possible	Access to the internal signal transition is not possible. User needs to use the external interfaces
Testing of the cores	The soft processor cores can be tested due to visibility of signals and as they are available in the form of the netlist	The hard processor core testing needs to be carried out using the stand-alone platform to verify the interface details and timing. Visibility to the internal signal is limited
Cost	High	Moderate
Power domains	Not efficient	Low-power architectures
Hardware software partitioning	The hardware software partitioning is possible using the soft processor cores	The hardware is fixed; hence, the software wrappers can be added to establish communication

Table 5.10 illustrates the difference between the processor hard and soft cores.

## 5.8 Important Takeaways and the Further Discussions

As discussed in this chapter, the following are few important points.

1. The processor cores are extensively used in the modern FPGAs.
2. The speed, data rate, and IO bandwidth are few of the important factors while developing the processor logic.
3. The architecture and micro-architecture of the design document should be used during the RTL design verification and implementation phase.
4. To improve the processor performance, use the pipelining by adding the pipelined control logic.
5. The pipelining stages can be used to improve the performance of the design.

6. The register balancing or register optimization can also be used to improve the design performance.
7. In the soft processor cores, the logic visibility during debug is higher as compared to that in the hard processor cores.
8. The multiprocessor architecture can be used in the design to improve the overall design features as it enables the multitasking.

The next chapter discusses the high-speed buses and protocols which is useful to understand the bidirectional buses, bus arbitration, and protocols.

# Chapter 6

## Buses and Protocols in SOC Designs



*The SOC performance is dependent on the speed of the buses and IO delays.*

**Abstract** The performance of the SOC is highly dependent on the bus architecture and arbitration schemes. This chapter discusses the few protocols used in the design and their use. The data transfer techniques between the SOC elements are discussed in this chapter. Even this chapter discusses bus architecture and data transfer schemes. The chapter is useful to understand the I2C, SPI, AHB bus protocols.

**Keywords** FIFO · SPI · I2C · UART · USART · AHB · Single bus  
Multiple bus architecture · Data rate · Latency · Throughput · Register

In all the SOC designs, we experience the use of the buses and the protocols. To transfer the data between the various SOC components, we use the FIFO, buffers, buses. The architecture and performance of such buses decides the overall performance of the design. The following section elaborates the need of the buses, protocols, and the high-speed architectures for the SOC designs.

### 6.1 Data Transfer Schemes

There are various mechanisms by using them the data can be exchanged between two computational elements. Following is the list of few mechanisms used to pass the data

1. Buses
2. Shared memory
3. FIFO
4. Write/read registers

5. On-chip network
6. Bus protocols

Let us try to discuss these mechanisms to understand the pros and cons.

1. **Buses:** As discussed earlier, the buses are used to transfer the data between two processing elements. The multiple bus configurations in the design of SOC can improve the overall design performance. Instead of using the single bus with higher clock frequency, it is always advisable to use the multiple buses to communicate with the faster and slower SOC processing elements. During the architecture evolution, we need to think about using the fast processor bus and slow peripheral bus.
2. **Shared Memories:** To transfer the large amount of data between the SOC processing elements, we can think of using the shared and dual-port memories. Consider the H.264 encoder, in such kind of the design the data packets or the video frames need to be transferred between two processing elements, and hence, it is essential to have shared memories.
3. **FIFO:** For interprocessor communication, one of the communication mechanisms is FIFO. During the design of the architecture, we can think of using the unidirectional FIFOs to transfer the data between the SOC processing elements. Depending on the data size, the FIFO depth can be chosen and it is not compulsory that the depth of the FIFOs should be same. But in the mechanism, the additional logic required to report the FIFO empty and FIFO full to the respective computational plane.
4. **Read/Write Registers:** If the small amount of the data need to be exchanged between different processors, then it is advisable to use read and write configuration registers. This mechanism is too simple as compared to use of the shared memories, FIFOs. So the better way is to use the point-to-point contact between the registers and should have the configuration and general purpose registers to perform the read or write operations.
5. **On-Chip Network:** The better mechanism for the large chunk of data transfer is to use the on-chip network. There is extensive amount of efforts to improve the on-chip network use in the design of SOC. This is out of scope for the discussion.
6. **Bus Protocols:** The bus protocols can be used to transfer the data between to processors or buses. The serial protocols like SPI, I2C, USB can be used to exchange the data in the form of packets from one of the computational elements to other and vice versa. The AHB, APB buses can be used to transfer the data between two computation elements. As protocols have the predefined architecture and the functionality, they can have the added advantage in the design.

The subsequent section discusses the design and implementation of the bus protocols, RTL design and verification and challenges in the design.

## 6.2 Tri-State Bus

The tri-state buses to avoid the contention of the data can be used in the design. At the top-level design, these buses can be added with the other SOC components. Depending on the design requirements, the unidirectional or bidirectional buses can be used in the design. Instead of using the tri-state logic, the multiplexer-based buses can be used for the design. The issue with the multiplexed buses is the long combinational paths and the delays.

The unidirectional tri-state 32-bit bus using the VHDL is described in Example 6.1.

The synthesis outcome of the example is shown in Fig. 6.1.

In the design depending on the need, the bidirectional or MUX-based buses can be used. As discussed, the SOC applications need to have the high-speed buses in the design to exchange the data between the processors and memories. The common

```
// Verilog code for the 32 bit tri-state bus

module tri_state_bus ( a_in, enable_in, y_out);

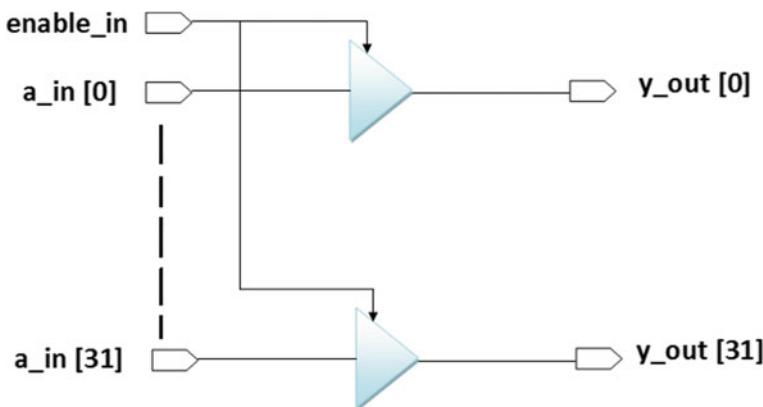
input [31:0] a_in,
input enable_in ;
output [31:0] y_out ;

reg [31:0] y_out;

always@(*)
begin
  if (enable_in)
    y_out = a_in;
  else
    y_out = 32'bzz;
end
endmodule
```

- The always block is sensitive to 'enable\_in', 'a\_in'.
- The 'y\_out' is assigned as a\_in for enable\_in ='1'
- For enable\_in ='0' y\_out is assigned as high impedance state.

**Example 6.1** Verilog code for the 32-bit tri-state bus



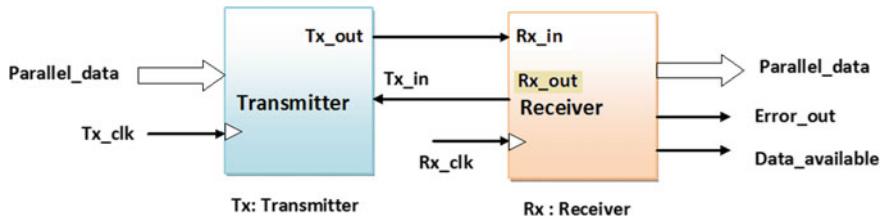
**Fig. 6.1** Tri-state 32-bit bus

shared bus needs the bus arbitration and the data exchanges become slower. Also, the issue of the data integrity needs to be addressed while exchanging the data between the processor and peripheral devices.

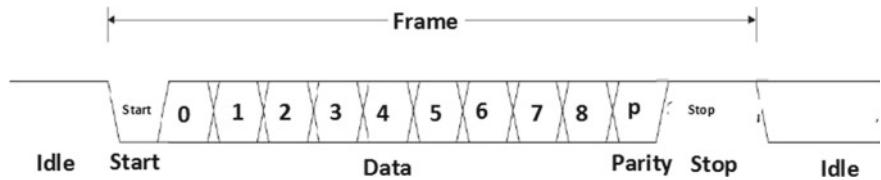
### 6.3 Serial Bus Protocols

In most of the SOC designs, we observe the need of the serial buses to transfer the data to the serial devices. Few of them frequently used to transfer data are USART, UART, Inter-Integrated Circuit (I2C), Serial Peripheral Interface (SPI), and other serial controllers.

1. **USART:** It is universal synchronous asynchronous transmitter and receiver, and its characteristics are explained below
  - a. Used for the synchronous or asynchronous serial communications.
  - b. By using the programmable features, the frequency for the transmission can be controlled. In other word, it has variable baud rate.
  - c. It supports the interrupt transmission control.
  - d. It supports the packet data of 5–9 bits with or without parity.
  - e. During transmission, the detection of error is possible.
2. **UART:** It is universal asynchronous receiver and transmitter, and few characteristics are described below
  - a. Used for the serial transmission of the data.
  - b. Data transfer is asynchronous in nature.
  - c. The interval between the data transfer is undefined.
  - d. It uses the start and end of packet for the serial transmission.

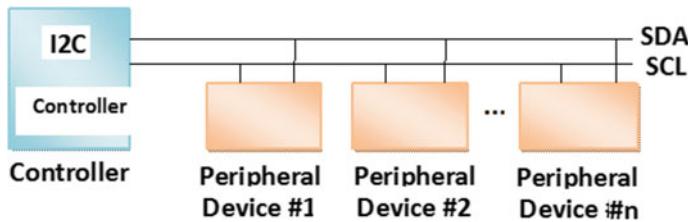


**Fig. 6.2** Serial data transfer using Rx, Tx

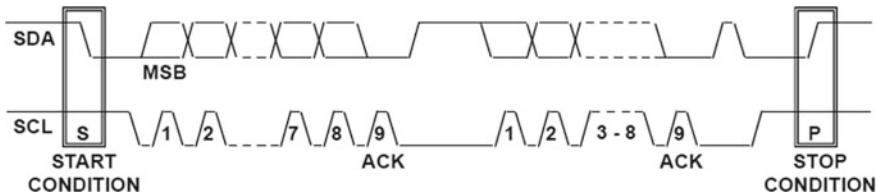


**Fig. 6.3** UART packet structure

- e. Baud rate is fixed and should be known to both sides for the transmission.
  - f. In the full duplex mode, the transmission and reception can be performed simultaneously.
3. Both sides can initiate the data transfer (Fig. 6.2).  
The single packet frame is shown in Fig. 6.3.  
As shown it starts with the start bit ‘logic 0’ and then 5–8 bits of the data and single parity bit. The frame ends with the 1 or 2 stop bits.  
The even parity is logic 0, and odd parity is logic 1.  
The receiver should know the following parameters:
- a. Baud rate that is programmable baud number
  - b. Number of bits for each frame
  - c. Parity
  - d. Number of stop bits
  - e. The transition detected from logic 1 to logic 0 indicates the start of the frame, and the state machine controller needs to detect this. Then during the next state at the middle edge it is essential to detect the data bits, parity, and stop bits.
  - f. Framing error: If zero is detected in the stop bits
  - g. Parity error: If the calculated parity does not correspond to the destination, then it generates the parity error.
4. **I2C Bus:** It is Inter-Integrated Circuit Bus and developed by the Philips semiconductor during the year 1980. The I2C is used when it is essential to communicate the devices occasionally, and the major advantage is that the addressing scheme. The addressing scheme allows the interconnection of the multiple devices without use of the additional wires. But there are limitations due to half-duplex mode



**Fig. 6.4** I2C bus controller



**Fig. 6.5** I2C timing sequence

and due to that not scalable for the larger number of devices. This is used for the control interface. Following are few of the characteristics

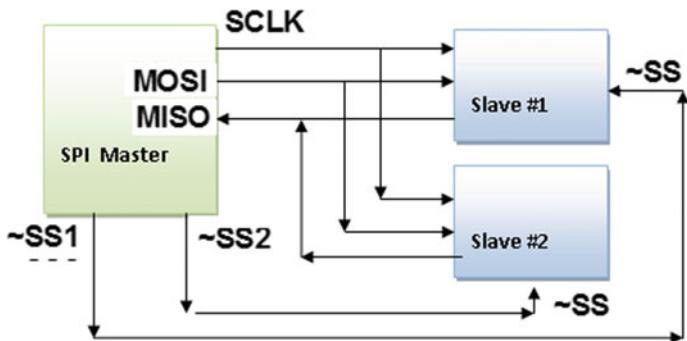
- The bus has inside bus length less than 1 m.
- The bus can be used for the effective serial communication for maximum distance of few meters.
- The I2C speed is 100 Kbps to 3.4 Mbps.
- I2C devices can have the separate data interface, for example, audio decoders, video decoders, etc. (Fig. 6.4).

The I2C has two wires, i.e., SDA and SCL. SDA is serial data and SCL is serial clock. It can be treated as half duplex serial master without any arbitration or chip select. The structure is equivalent to wire AND. Logic '1' connection to these lines indicates the pull-up resistor. Logic 0 indicates the open drain configuration. The I2C timing sequence is shown in Fig. 6.5

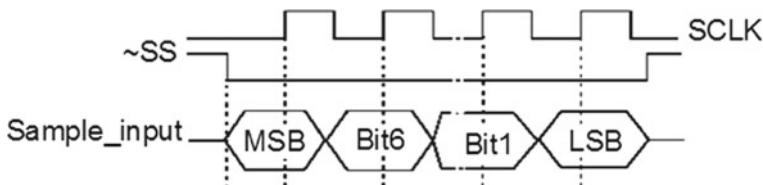
### How the communication is established in I2C?

The timing sequence is shown in Fig. 6.5 and as shown for serial data transfer following is the sequence:

- **Start (S):** Falling edge when SCL is logic '1',
- **ACK:** the receiver pulls down the status of SDA to logic '0', and transmitter maintains SDA to logic '1',
- **Stop (P):** Rising edge on the SDA when SCL is logic '1'. Data on SDA is valid when SCL is logic '0' and SCL is equal to logic '1' for the valid transmission.
- Master sends the start signal (S) and the clk is generated on the SCL line.
- Master sends the 7-bit slave address.



**Fig. 6.6** SPI bus master–slave interface

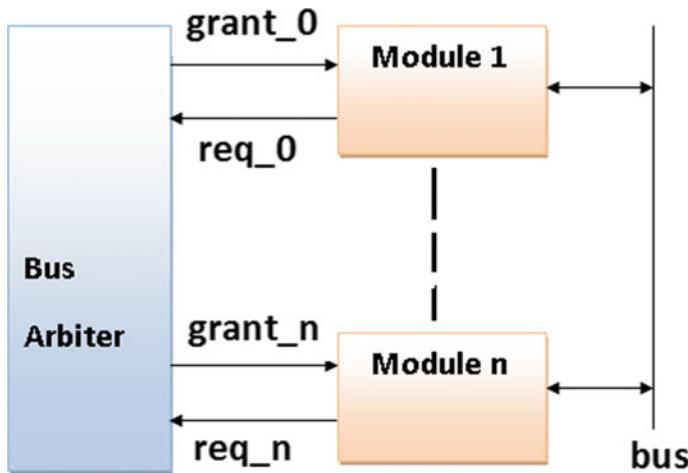


**Fig. 6.7** SPI timing diagram

- Master sends the data bit read or write(R/W) signal. R/W is equal to logic ‘0’ indicates the slave will receive the data, and logic ‘1’ indicates the slave will transmit the data.
- After this transmitter, either slave or master sends the acknowledge (ACK) bit.
- Then the transmitter sends the 8 bit of data.
- After receiving the byte, the receiver sends an acknowledge (ACK).
- For burst of the data bytes, the controller needs to repeat the step 5 and step 6.
- For the write transaction; if the master is transmitter, then sends the stop (P) after the last byte of data.
- For the read transaction; if the master is receiver, then it does not send the ACK signal but only sends the stop (P) signal to confirm the end of the transmission.

5. **SPI BUS:** It is Serial Peripheral Interface bus and synchronous in nature. The communication is established between the master–slave devices. It has 4 lines, 2 data lines MOSI: Master data output, slave data input, and MISO: Master data input and slave data output. The two control lines are SCLK: clock and complement of SS: Slave select (Figs. 6.6 and 6.7).

Active edge of the clock is determined by the two parameters, and they are clock polarity (CPOL) and clock phase (CPHA). Both are logic zero or both are logic one indicates the rising edge. Both parameters are not equal and then indicate the falling edge. Care should be taken that the master and slave should be configured with the same set of parameters; otherwise, they will not communicate.



**Fig. 6.8** Bus arbitrations

## 6.4 Bus Arbitration

The shared bus can be shared by multiple functional modules (components) in the design environment. Depending on the request generated by one of the modules, the bus can be granted if it is not busy.

The bus arbitration is used to sample the requests generated by the different SOC components sharing the common bus and to grant the request to one of the SOC components.

As shown in Fig. 6.8 the multiple components (module 1 to module  $n$ ) generate the request to the bus arbiter and wait for the grant signals from the arbiters. After receiving the grant signal from the bus arbiter, one of the components gains the control of the shared bus.

In the practical environment, there are many schemes to design the bus arbitration, and they can be daisy chaining, round-robin, static arbitrations. Depending on the design requirements, these schemes can be used in the SOC designs.

## 6.5 Design Scenarios

The design scenarios encountered during the parallel data transfer and the serial data transfer are discussed in this section.

```

module static_arbitration ( clk, reset_n, request_0, request_1, request_2,
                           grant_0,grant_1,grant_2);

  input clk;
  input reset_n;
  input request_0, request_1,request_2;
  output reg grant_0, grant_1,grant_2;

  always @((posedge clk or negedge reset_n)
begin
  if (~reset_n)
    {grant_2,grant_1,grant_0}<=3'b000;           ←-----|
  else
    begin
      grant_0<=request_0;
      grant_1 <= (request_1 && (!request_0));
      grant_2<=(request_2&& (!request_1||request_0));
    end
  end
endmodule

```

- In this the request\_0 has the highest priority and request\_2 has the lowest priority.

**Example 6.2** Synthesizable Verilog code of the static arbiter

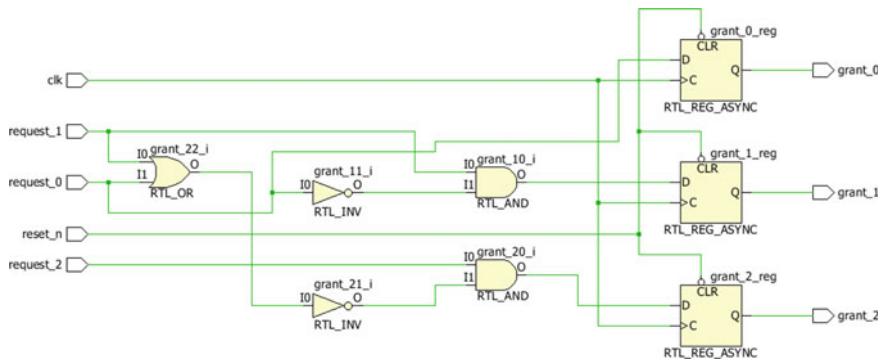
### 6.5.1 Scenario 1: Static Arbitration

The static arbitration using Verilog is described in Example 6.2.

The HDL synthesis outcome is shown in Fig 6.9.

### 6.5.2 Scenario 2: Bidirectional Data Transfer and Registered IOs

The bidirectional buses are used in the design to transfer the data to/from the processor. The Verilog code is described in Example 6.3 and Fig. 6.10.



**Fig. 6.9** Synthesis result of static arbiter

```

module bidirectional_bus (data_to_bus, send_data, receive_data, data_from_bus,
qout);

parameter N = 16;

input send_data;

input receive_data;

input [N-1:0] data_to_bus;

output [N-1:0] data_from_bus;

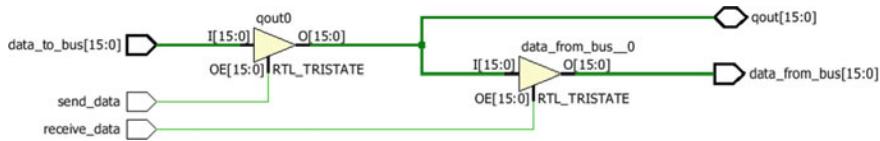
inout [N-1:0] qout;

wire [N-1:0] qout, data_from_bus;
    assign data_from_bus = receive_data ? qout : {N{1'bz}};
    assign qout = send_data ? data_to_bus : {N{1'bz}};
endmodule

```

- It infers the bidirectional 16 bit bus.
- receive\_data and send\_data are used to control the data transfer direction

**Example 6.3** Synthesizable Verilog code for bidirectional bus



**Fig. 6.10** Synthesis outcome for the bidirectional IO

### 6.5.3 Scenario 3: UART Transmitter and Receiver Design

The following section describes the design of the UART transmitter and receiver. The design of baud rate generator, receiver and transmitter section with associated logic is described in this section (Examples 6.4, 6.5, 6.6, and 6.7).

## 6.6 High-Density FPGA Fabric and Buses

Most of the high-density FPGAs like Xilinx and Intel uses the transceivers and other high-speed bus transfer interfaces and can be used during the SOC design.

### 6.6.1 Xilinx-7 Series Transceivers

The architecture has the low-power gigabit transceiver. Due to low-power architecture, the chip-to-chip interface is optimized and this is one of the powerful features of this FPGA. The high-performance transceiver is capable to support the data rate from 6.6 to 28.05 Gb/s depending on the device family of the Virtex-7 FPGA.

The transceiver count is 16 in the Artix-7 FPGA family, up to 32 transceivers in the Kintex-7 FPGA family, and up to 96 transceivers in the Virtex-7 FPGA family.

To improve the IP portability, the architecture of the serial transceiver uses the ring oscillators and LC tank circuit. The transmitter and receiver circuits are different, and they use the PLL to multiply the reference clock by the programmable number up to 100 to get the bit serial clock.

#### 6.6.1.1 Transmitter

##### Following are the key features of gigabit transmitter:

1. The transmitter is parallel to serial converter with conversion ratio of 16, 20, 32, 40, 64, or 80.
2. The GTZ transmitter supports up to 160-bit data width.
3. It uses TXOUTCLK used to register the parallel data.

```
//The code describes the baud rate generator for the UART receiver
module baud_gen ( clk, reset_n, max_tick_size, q_out);
parameter N= 4;
parameter Y:= 10;

input clk, reset_n;
output max_tick_size;
output [N-1:0] q_out ;
wire [N-1:0] q_out;

reg [N-1:0] tmp_reg;
reg [N-1:0] tmp_next;

always @ (posedge clk or negedge reset_n)
begin
if (reset_n)
tmp_reg <= 0;
else
tmp_reg <= tmp_next;
end

//next state logic is described below
always@ ( tmp_reg)
begin
if (tmp_reg==(Y-1))

tmp_next<= 0;
max_tick_size<=1'b0;

else
tmp_next<= tmp_reg + 1;

max_tick_size<=1'b1;

end

endmodule
```

- Depending on the operating frequency of the design the frequency can be chosen.
- The tick size is equal to maximum frequency/no of samples per second.

#### Example 6.4 Synthesizable Verilog code of baud rate generator

4. The incoming parallel data is fed through an optional FIFO and to provide the sufficient number of transitions, it has additional support of 8B/10B, 64B/66B encoding schemes.
5. The output of these transmitters drives the PC board with the single-channel differential output signal.

```

//The code describes the UART receiver
module uart_receiver (clk, reset_n, receiver_in, baud_tick, receiver_done_tick, receiver_data_out);
parameter data_width = 8;
parameter baud_rate_tick= 16;

input clk, reset_n;
input receiver_in;
input baud_tick;
output reg receiver_done_tick;
output [data_width-1:0] receiver_data_out;

wire [data_width-1:0] receiver_data_out;

parameter idle=2'b00;
parameter start=2'b01;
parameter data=2'b10;
parameter stop=2'b11;
reg [1:0] state_reg, state_next: state_type;
reg [3:0] s_reg, s_next;
reg [2:0] n_reg, n_next;
reg [7:0] b_reg, b_next;

always@ (posedge clk or negedge reset_n)
begin
if ~(reset_n)
state_reg <= idle;
s_reg <= 0;
n_reg <= 0;
b_reg <= 0;
else
state_reg <= state_next;
s_reg <= s_next;
n_reg <= n_next;
b_reg <= b_next;
end
//description of the next state logic
always@ (state_reg, s_reg, n_reg, b_reg, baud_tick, receiver_data)
begin
state_next <= state_reg;
s_next <= s_reg;
n_next <= n_reg;
b_next <= b_reg;
receiver_done_tick <= 0;
case state_reg
idle :
if (~receiver_in)
state_next <= start;
s_next <= 0;
b_reg <= b_next;
end

```

- The receiver logic uses the output of the baud rate generator and using the UART protocol it samples the serial input.
- The receiver data out is 8 bit parallel output.

#### Example 6.5 Synthesizable Verilog code of UART receiver

6. To compensate for the PC board losses, the output signal pair has programmable signal swing.
7. To reduce the power consumption, this swing can be reduced for the shorter channels.

```

// next state logic continued for the receiver section
always@ (state_reg, s_reg, n_reg, b_reg, baud_tick, receiver_in)
begin
    state_next <= state_reg;
    s_next <= s_reg;
    n_next <= n_reg;
    b_next <= b_reg;
    receiver_done_tick <= '0';
    case state_reg is
        idle:
            if(~rx)
                state_next <= start;
                s_next <= 0;
                n_next <= n_reg + 1;
            else
                s_next <= s_reg + 1;

        stop :
            if(s_tick)
                if(s_reg == (Baud_rate_tick-1))
                    state_next <= idle;
                    receiver_done_tick <= 1;
                else
                    s_next <= s_reg + 1;
            endcase;
            end
        assign receiver_data_out = b_reg;
    endmodule

```

➤ Depending of the baud rate tick the receiver data output is assigned.

#### **Example 6.5 (continued)**

##### **6.6.1.2 Receiver**

###### **Following are key features of gigabit receivers:**

1. The receiver is serial to parallel converter with conversion ratio of 16, 20, 32, 40, 64, or 80.
2. The GTZ receiver supports up to 160-bit data width.
3. To guarantee sufficient data transition, it uses non-return-to-zero (NRZ) encoding.
4. The parallel data is transferred into the FPGA using the RXUSRCLK.
5. For short channels to reduce power consumption by almost 30%, the transceiver offers special low-power (LPM) mode.

```
//external logic interface Verilog description
module interface_logic (clk, reset_n, clr_flag, set_flag,data_in, data_out,status_out);

parameter N= 8;
input clk, reset_n;
input clr_flag, set_flag;
input [N-1:0] data_in;
output [N-1:0] data_out;
output status_flag;

reg [N-1:0] buf_reg, buf_next;
reg flag_reg, flag_next;

always@ (posedge clk or negedge reset_n)

begin
if (!reset_n) then
buf_reg <=0;
flag_reg <= 0;
else
buf_reg <= buf_next;
flag_reg <= flag_next;

end
// next-state logic for the interface logic
always@ (buf_reg, flag_reg, set_flag, clr_flag, data_in)
begin
buf_next <= buf_reg;
flag_next <= flag_reg;
if (set_flag)
buf_next <= data_in;
flag_next <= 0;
else if (clr_flag)
flag_next <= 0;
end
//assignment to the output
assign data_out <= buf_reg;
assign status_flag <= flag_reg;

endmodule
```

➤ The logic is used to interface with the eternal interfaces

**Example 6.6** Synthesizable Verilog code of external interface logic

### 6.6.2 Intel FPGA Transceivers

The Intel FPGA transceiver block is shown in Fig. 6.11.

The Stratix 10 Intel FPGA features and capabilities are listed in Table 6.1.

As shown most of the high-density FPGAs nowadays has the high-speed interfaces and supports the standard protocols.

```
//transmitter code verilog description

module uart_txasmitter (clk, reset_n, transmitter_start, baud_tick, data_in,
transmit_done_tick, transmit_out);
parameter data_width = 8;
parameter Baud_rate_tick = 16;
input clk, reset_n;
input transmitter_start;
input baud_tick;
input [data_width-1:0] data_in;
output transmit_done_tick;
output transmit_out;
parameter idle=2'b00;
parameter start=2'b01;
parameter data=2'b10;
parameter stop=2'b11;
reg state_reg, state_next: state_type;
reg [3:0] s_reg, s_next;
reg [2:0] n_reg, n_next;
reg [7:0] b_reg, b_next;
reg tx_reg, tx_next;

always @ (posedge clk or negedge reset_n)
begin
if(~reset_n)
state_reg <= idle;
s_reg <=0;
n_reg <= 0;
b_reg <= 0;
tx_reg <= 1;
else
state_reg <= state_next;
s_reg <= s_next;
n_reg <= n_next;
b_reg <= b_next;
tx_reg <= tx_next;
end

// the next state logic description is given below
always@ (state_reg, s_reg, n_reg, b_reg, s_tick,
tx_reg, transmitter_start, data_in)
begin
state_next <= state_reg;
s_next <= s_reg;
n_next <= n_reg;
b_next <= b_reg;
tx_next <= tx_reg;
```

➤ The logic is used to generate the serial data depending on the baud clock rate.

**Example 6.7** Synthesizable Verilog code of UART transmitter

```
transmit_done_tick <= 1'b1;
case state_reg
  idle :
    tx_next <= 1'b1;
    if(transmitter_start)
      state_next <= start;
      s_next <= 0;
      b_next <= data_in;
    start :
      tx_next <= 1'b0;
      if(s_tick)
        if(s_reg == 15)
          state_next <= data;
          s_next <= 0;
          n_next <= 0;
        else
          s_next <= s_reg + 1;

      data :
        tx_next <= b_reg[0];
        if(s_tick)
          if(s_reg == 15)
            s_next <= 0;
            b_next <= {0, b_reg[7: 1]};
            if(n_reg == (data_width - 1))
              state_next <= stop;
            else
              n_next <= n_reg + 1;

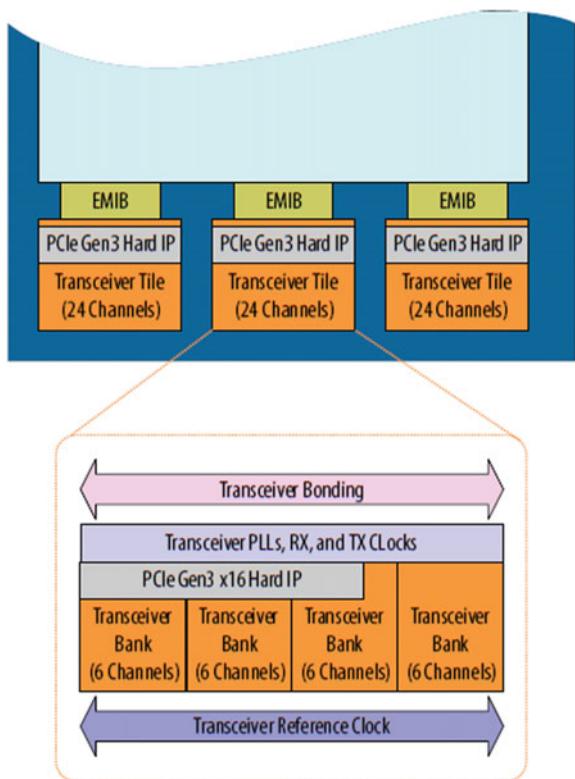
        else
          s_next <= s_reg + 1;

      stop :
        tx_next <= 1'b1;
        if(s_tick == '1')
          if(s_reg == (baud_rate_tick-1))
            state_next <= idle;
            transmit_done_tick <= 1'b1;
          else
            s_next <= s_reg + 1;
        endcase
      end
      assign transmit_out = tx_reg;
endmodule
```

➤ Depending on the baud rate tick the transmitter output is assigned.

**Example 6.7** (continued)

**Fig. 6.11** Intel FPGA transceiver [1]



## 6.7 Single Master AHB

The single master AHB-lite bus architecture is shown in Fig. 6.12. The master can generate the address and can be decoded by the decoder. Decoder generates the select signal to select the slave.

The write and read transactions can be initiated by using such type of architecture. For more details, refer the AHB/APB architectures and ARM processor system.

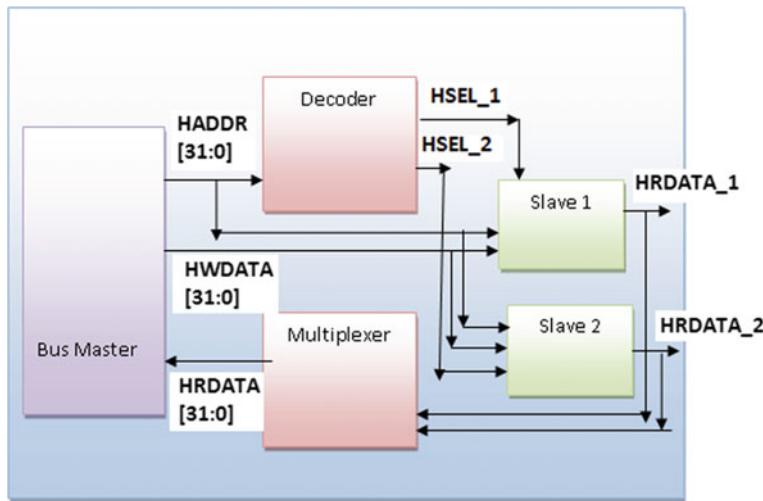
## 6.8 How This Discussion Is Useful During SOC Prototyping?

If we see the need in the most of the prototyping system, then we can think of following:

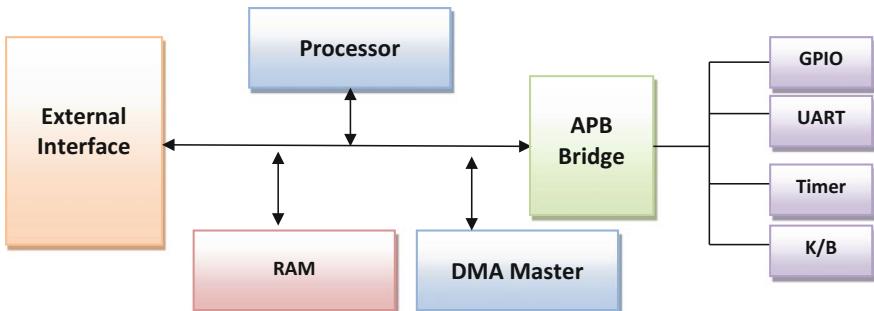
- Need of the processor buses and peripheral buses.
- In such context, it is better to use the AHB and APB buses.

**Table 6.1** Intel Stratix 10 FPGA transceiver features [1]

Feature	Capability
Chip-to-chip data rates	1 Gbps to 28.3 Gbps (Intel Stratix 10 GX/SX devices)
Backplane support	Drive backplane at data rates up to 28.3 Gbps, Including 10GBASE-KR compliance
Optical module support	SFP+/SFP, XFP, CXP, QSFP/QSFP28, QSFPDD, CFP/CFP2/CFP4
Cable driving support	SFP+ Direct Attach, PCI Express over cable, eSATA
Transmit pre-emphasis	5-tap transmit pre-emphasis and de-emphasis to compensate for system channel loss
Continuous-time linear equalizer (CTLE)	Dual mode, high-gain, and high-data rate, linear receive equalization to compensate for system channel loss
Decision feedback equalizer (DFE)	15 fixed tap DFE to equalize backplane channel loss In the presence of crosstalk and noisy environments
Advanced digital adaptive parametric tuning (ADAPT)	Fully digital adaptation engine to automatically adjust all link equalization parameters including CTLE, DFE, and VGA blocks—that provide optimal link margin without Intervention From user logic
Precision signal integrity calibration engine (PreSICE)	Hardened calibration controller to quickly calibrate all transceiver control parameters on power-up, which provides the optimal signal Integrity and jitter performance
ATX transmit PLLs	Low jitter ATX (Inductor–capacitor) transmit PLLs with continuous tuning range to cover a wide range of standard and proprietary protocols, with optional fractional frequency synthesis capability
Fractional PLLs	On-chip fractional frequency synthesizers to replace onboard crystal oscillators and reduce system cost
Digitally assisted analog CDR	Superior jitter tolerance with fast lock time
On-die instrumentation-eye viewer and jitter margin tool	Simplify board bring-up, debug, and diagnostics with non-intrusive, high-resolution eye monitoring(Eye Viewer), Also inject jitter from transmitter to test link margin in system
Dynamic reconfiguration	Allows for independent control of each transceiver channel Avalon memory-mapped interface for the most transceiver flexibility
Multiple PCS-PMA and PCS-core to FPGA fabric interface widths	8-, 10-, 16-, 20-, 32-, 40-, or 64-bit interface widths for flexibility of deserialization width, encoding, and reduced latency



**Fig. 6.12** AHB-lite single master–multiple slave systems



**Fig. 6.13** AHB-APB bus use in the design

The bus arbiters for the multiple master and multiple slave interfaces.

- d. The environment should have the IO devices interfacing using the APB bus.
- e. The APB bridge can be used to establish the communication with the AHB bus.

Most of the high-density FPGAs have all such kind bus interfaces and can be used to communicate between multiple IO devices with other SOC components (Fig. 6.13).

## 6.9 Important Takeaways and Further Discussions

The following are few important points to conclude this chapter

1. The buses in the design are used to exchange the data between the processing elements.
2. The bus width and the data exchange speed decided the overall design performance.
3. The predefined functional- and timing-proven bus architectures can be used during the SOC prototyping to have the improved design performance.
4. The I2C, SPI, USB can be used to transfer the data between the SOC and other system.
5. The high-speed AHB and APB buses can be used in the architecture.
6. To avoid the bus contentions, use the arbitration scheme in the SOC design.

The next chapter discusses the memory and memory controllers in the design.

## Reference

1. [www.altera.com](http://www.altera.com)

# Chapter 7

## Memory and Memory Controllers



*The double data rate memory and the constraints at the interface boundary decide the overall data transfer speed.*

**Abstract** In the SOC designs, the transfer of the data from the external memories needs the dedicated memory controller. The SDRAM or DDR memory controllers are used extensively in the SOC designs. The available IPs of such kind of controllers can be integrated with other SOC components. During prototyping, it is essential to have the FPGA equivalent logic of such IP cores. By considering all above, the chapter discusses the memory controllers and their interfaces with the external memory. The timing constraints for such type of controller are decisive factor for the overall design and are discussed in this chapter.

**Keywords** SOC · Memory controllers · SDRAM · DDR · AHB · APB · Latency Max and min delay · Command · Address · Data · Clk · Generated clock · Timing · Setup · Hold · Hard core · Soft core

During the SOC prototyping, the design of the memory controller and integration with the other SOC components to achieve desired speed is the important aspect. The SOC RTL of such memory controller is not compatible with the FPGA equivalent and needs the modifications. Under such circumstances, the RTL tweaks can be used. But the better approach is to use the memory IPs and use the interface wrappers to communicate with the eternal memories. The SDRAM or DDR interface timing is crucial, and it is bottleneck to achieve the read and write cycle timing. The following few sections discuss the architecture and interfaces for such kind of designs.

## 7.1 Memory

In the SOC designs, the data need to be stored. The stored data can be used by the processing unit to perform the operation. The memories can be classified as

1. Internal memory
  - a. Distributed RAM
  - b. Single-port RAM
  - c. Dual-port RAM
2. External memory
  - a. SRAM
  - b. SDRAM
  - c. DDR

This section describes the RTL design for the distributed RAM, single-port RAM, and dual-port RAM.

### 7.1.1 *Dual-Port Distributed RAM*

The distributed RAM frequently referred using the Verilog RTL is described in Example 7.1 (Fig. 7.1).

The device utilization for the Xilinx xc7v585tffg1157-3 is shown in the following snapshot (Fig. 7.2).

### 7.1.2 *Single-Port RAM*

The single-port RAM with the read first mode, write first mode, and no change mode is described in this section.

#### 7.1.2.1 *Single-Port RAM (No Change Mode)*

The single-port RAM using Verilog RTL is described in Example 7.2 (Figs. 7.3 and 7.4).

#### 7.1.3 *Single-Port RAM (Read First Mode)*

The single-port RAM using Verilog RTL is described in Example 7.3.

### 7.1.4 Single-Port RAM (Write First Mode)

The single-port RAM with the write first mode is described in Example 7.4.

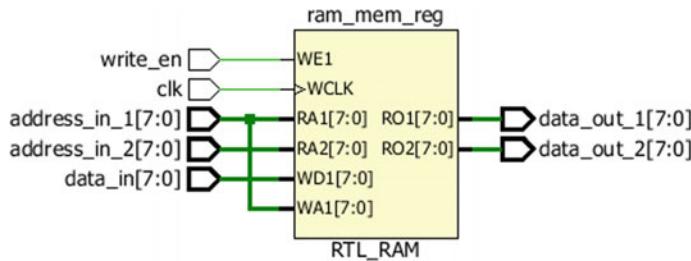
### 7.1.5 Dual-Port RAM

The dual-port RAM with read first mode with the two write ports is described using Verilog RTL in Example 7.5 (Figs. 7.5 and 7.6).

```
module distributed_ram (clk, write_en, address_in_1, address_in_2, data_in, data_out_1,
data_out_2);

input clk;
input write_en;
input [7:0] address_in_1;
input [7:0] address_in_2;
input [7:0] data_in;
output [7:0] data_out_1;
output [7:0] data_out_2;
reg [7:0] ram_mem [255:0];
always @(posedge clk)
begin
if (write_en)
ram_mem[address_in_1] <= data_in;
end
assign data_out_1 = ram_mem[address_in_1];
assign data_out_2 = ram_mem[address_in_2];
endmodule
```

**Example 7.1** Verilog RTL of dual-port distributed RAM



**Fig. 7.1** Synthesis result of distributed RAM

**Fig. 7.2** Snapshot of device utilization

Resource	Estimation	Available	Utilization %
LUT	74	364200	0.02
Memory LUT	64	111000	0.06
I/O	42	600	7.00
BUFG	1	32	3.12

## 7.2 Double Data Rate Memory

In most of the SOC designs, we need to have the memory controllers to transfer the data at high speed. Let us consider the transfer of the 16 bits of the data. If the controller 8-bit of data transfer mechanism and works using the single clock cycle, then the two clock cycles are required to transfer the 16 bits of the data. To speed up the data transfer, we can think of design of the controller which can transfer lower byte on the rising edge of the clock and higher byte on the falling edge of the clock. Effectively, the half cycle data transfer. The real challenge in such type of the design is to align the clock phases while transferring the data. The constraints and the clocking mechanism play the crucial role in such type of the design (Fig. 7.7).

The lower byte and higher byte of data can be sampled on the rising edge and falling edge of the clock, respectively. Mechanism to sample the DQ on the active edge of the clock is shown in Fig. 7.8. The timing relationship between the DQS and DQ is shown in Fig. 7.9. The designer should take care that the data can be sampled at the middle of the active edge.

## 7.3 SRAM Controllers and Timing Constraints

If we consider the design of the SRAM controller, then we can have the important functional blocks as

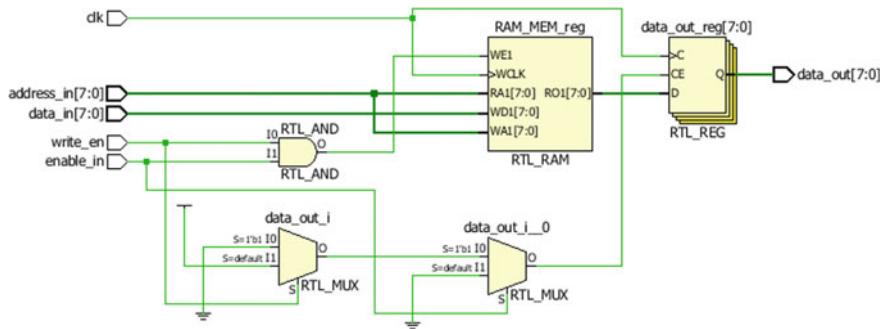
1. Command generator
2. Data access interface
3. Address and control logic

```
module single_port_RAM (clk, address_in, write_en, enable_in, data_in, data_out);  
    input clk;  
    input [7:0] address_in;  
    input write_en;  
    input enable_in;  
    input [7:0] data_in;  
    output [7:0] data_out;  
    reg [7:0] data_out;  
    reg [7:0] RAM_MEM [255:0];  
    always @(posedge clk)  
    begin  
        if (enable_in)  
            begin  
                if (write_en)  
                    begin  
                        RAM_MEM[address_in] <= data_in;  
                    end  
                else  
                    data_out <= RAM_MEM[address_in];  
            end  
        end  
    endmodule
```

**Example 7.2** Verilog RTL for single-port RAM for read first and write first mode

Figure 7.10 describes these functional blocks to generate the interface signal.

The SRAM controller interfaced with the SRAM is shown in Fig. 7.11. The interface signal description is shown in Table 7.1.



**Fig. 7.3** Synthesis result for the single-port RAM

Resource	Estimation	Available	Utilization %
I/O	27	600	4.50
BRAM	0.5	795	0.06
BUFG	1	32	3.12

**Fig. 7.4** Device utilization snapshot

To specify the constraints, use the following steps

1. Define the primary clock.
2. Define the generated clock from PLL.
3. Define the address and control interface constraints.
4. Define the constraints for the data output.
5. Define the constraints for the data input.

```
# The following can be sample script using Synopsys constraints
#1
create_clock -name PLL_CLK -period 10 [get_pins UPPL/CLK_OUT]
#2
create_generated_clock -name clk -source [get_pins UPPL/CLK_OUT]
-divide_by 1 [get_ports/clk]
#3
set_output_delay -max 2.5 -clock clk [get_ports ADDR]
set_output_delay -min 1.0 -clock clk [get_ports ADDR]
```

```
#4  
set_output_delay -max 3.0 -clock clk [get_ports DQ]  
set_output_delay -min 1.2 -clock clk [get_ports DQ]  
#5  
set_input_delay -max 4.0 -clock clk [get_ports DQ]  
set_input_delay -min 1.5 -clock clk [get_ports DQ]
```

```
module single_port_RAM (clk, address_in, enable_in, write_en, data_in, data_out);  
  
input clk;  
  
input [7:0] address_in;  
  
input write_en;  
  
input enable_in;  
  
input [7:0] data_in;  
  
output [7:0] data_out;  
  
reg [7:0] RAM_MEM [255:0];  
  
reg [15:0] data_out;  
  
always @(posedge clk)  
begin  
  
if (enable_in)  
begin  
  
if (write_en)  
  
RAM_MEM[address_in]<=data_in;  
  
data_out <= RAM_MEM[address_in];  
  
end  
  
end  
  
endmodule
```

**Example 7.3** Verilog RTL of single-port RAM with read first mode

```
module single_port_RAM (clk, address_in, write_en, enable_in, data_in, data_out);  
    input clk;  
    input [7:0] address_in;  
    input write_en;  
    input enable_in;  
    input [7:0] data_in;  
    output [7:0] data_out;  
    reg [7:0] data_out;  
    reg [7:0] RAM_MEM [255:0];  
    always @(posedge clk)  
    begin  
        if (enable_in)  
            begin  
                if (write_en)  
                    begin  
                        RAM_MEM[address_in] <= data_in;  
                        data_out <= data_in;  
                    end  
                else  
                    data_out <= RAM_MEM[address_in];  
            end  
        end  
    endmodule
```

**Example 7.4** Verilog RTL of single-port RAM with write first mode

```

module dual_port_1
(clk_1,clk_2,enable_in_1,enable_in_2,write_en_1,write_en_2,address_in_1,address_in_2,data_in_1,
data_in_2,data_out_1,data_out_2);
input clk_1,clk_2;
input enable_in_1,enable_in_2;
input write_en_1,write_en_2;
input [7:0] address_in_1,address_in_2;
input [7:0] data_in_1,data_in_2;
output [7:0] data_out_1,data_out_2;
reg [7:0] data_out_1,data_out_2;
reg [7:0] ram_mem [255:0];
always @(posedge clk_1)
begin
if(enable_in_1)
begin
if(write_en_1)
ram_mem[address_in_1] <= data_in_1;
data_out_1 <= ram_mem[address_in_1];
end
end
always @(posedge clk_2)
begin
if(enable_in_2)
begin
if(write_en_2)
ram_mem[address_in_2] <= data_in_2;
data_out_2 <= ram_mem[address_in_2];
end
end
endmodule

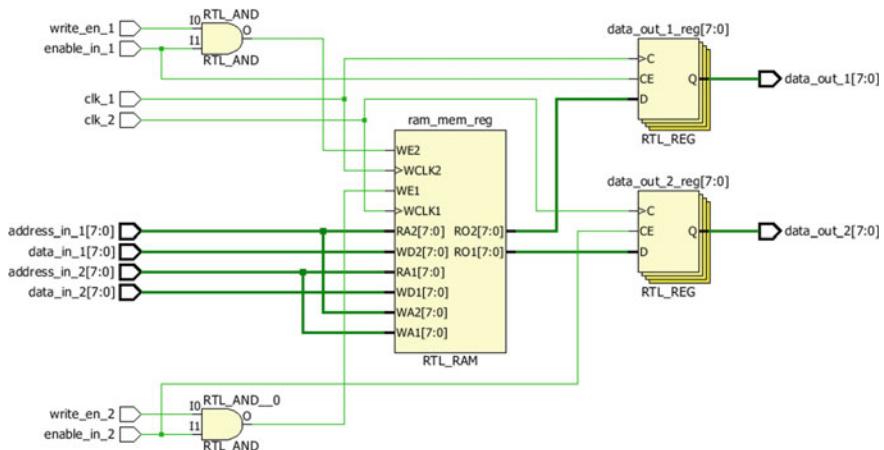
```

**Example 7.5** Verilog RTL of dual-port RAM

## 7.4 SDRAM Controller and Timing Constraints

The SDRAM controller is interfaced with the SDRAM. The CAC, DQ, DQS, and clk are important interface signals shown in Fig. 7.12.

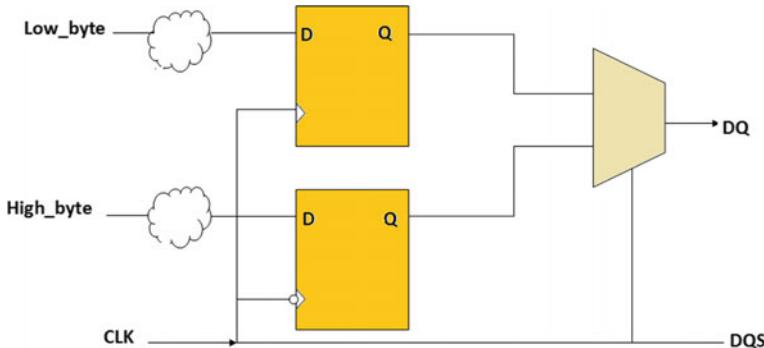
The interface signal description is shown in Table 7.2.



**Fig. 7.5** Synthesis result for the dual-port RAM

Resource	Estimation	Available	Utilization %
I/O	54	600	9.00
BRAM	0.5	795	0.06
BUFG	2	32	6.25

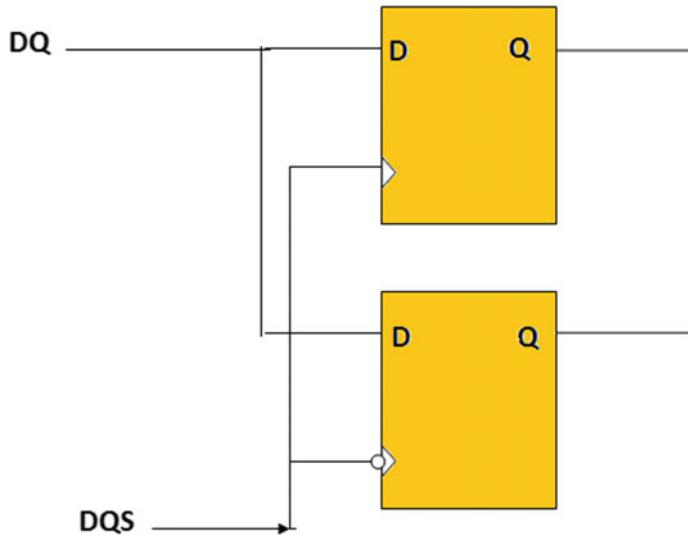
**Fig. 7.6** Device utilization for the Virtex-7 family device



**Fig. 7.7** DQ and DQS generation logic

To specify the constraints, use the following steps

1. Define the primary clock.
2. Define the generated clock from PLL.
3. Set the output constraints for CAC.
4. Define the input constraints for the rising clock edge.



**Fig. 7.8** Capturing the lower and upper byte of data on active edge of DQS

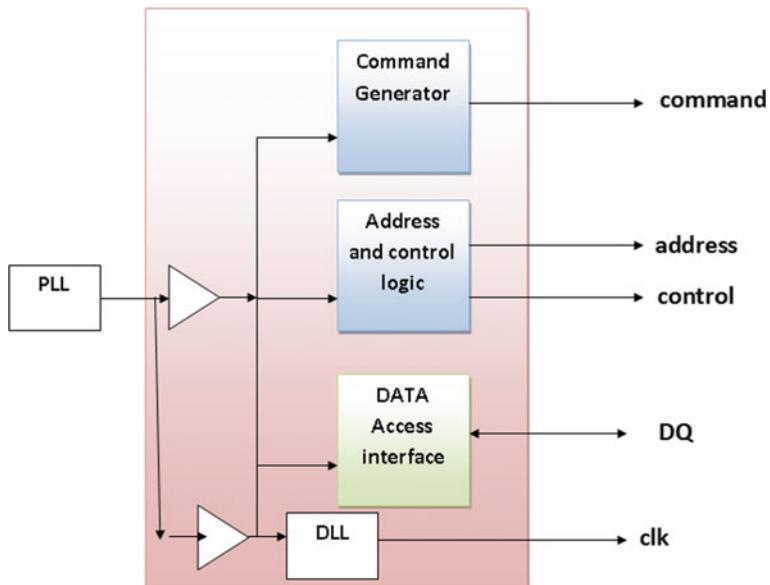


**Fig. 7.9** Timing relationship of DQS and DQ

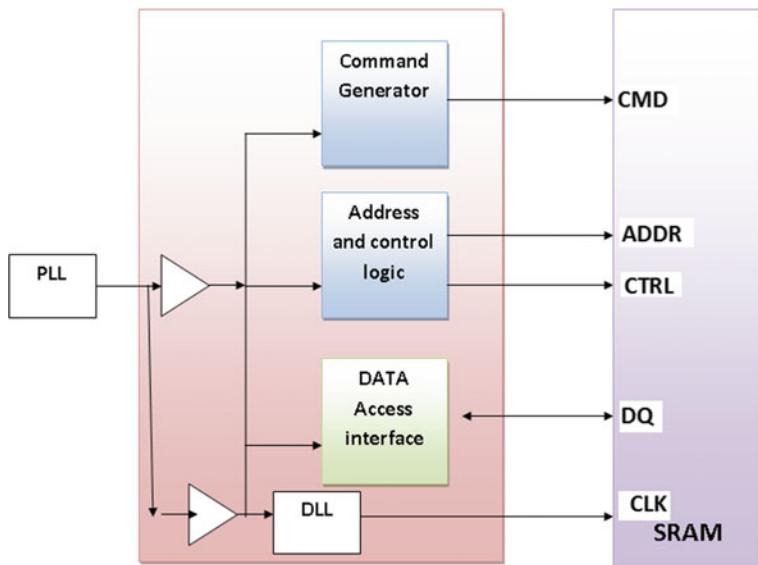
5. Define the input constraints for the falling clock edge.
6. Launch the data and capture the data on edge.

*The following can be sample script using Synopsys constraints*

```
#1
create_clock -name PLL_CLK -period 5 [get_pins UPPL/CLK_OUT]
#2
create_generated_clock      -name      clk_DDR      -source      [get_pins
UPPL/CLK_OUT] -divide_by 1 [get_ports/clk_DDR]
#3
set_output_delay -max 1-clock clk_DDR [get_ports CAC]
set_output_delay -min -1-clock clk_DDR [get_ports CAC]
```



**Fig. 7.10** Important SRAM interface signals



**Fig. 7.11** SRAM controller interfacing with SRAM

**Table 7.1** SRAM interface signal description

Interface_signal	Description
CMD	This interface signal carries the command information
ADDR	These interface signals are used to carry the address
CNTRL	The control signal information is carried by these signals
DQ	Bidirectional data bus
CLK	Clock for the SRAM interface

```

#4
set_input_delay -max -1.0 -clock DQS [get_ports DQ]
set_input_delay -min -0.5 -clock DQS [get_ports DQ]
#5
set_input_delay -max 0.4 -clock DQS -clock_fall [get_ports DQ]
set_input_delay -min -0.4 -clock DQS -clock_fall [get_ports DQ]
#6
Set_multicycle_path o -setup -to UPFF/D

```

While specifying the constraints for the write, use the clock multiplied by one.

```

create_clock -name PLL_CLKX1 -period 7 [get_ports CLKX1]
#2
create_generated_clock -name DQS -source CLKX1 -edges {1 2 3}
-edge_shift {1.7 1.7 1.7} [get_ports DQS]
#3
set_output_delay -max 0.25 -clock DQS [get_ports DQ]
set_output_delay -max 0.3 -clock DQS -clock_fall [get_ports DQ]
#4
set_output_delay -min 0.2 -clock DQS [get_ports DQ]
set_output_delay -max -0.3 -clock DQS -clock_fall [get_ports DQ]

```

## 7.5 FPGA Design and Memories

BRAM is embedded memory, and the FPGA BRAM can be configured as single-port and dual-port BRAM. Depending on the architecture of FPGA device, each BRAM consists of the number of static RAM cells. Among them, the few cells are used for the configuration of the memory and remaining are used for the data storage. The

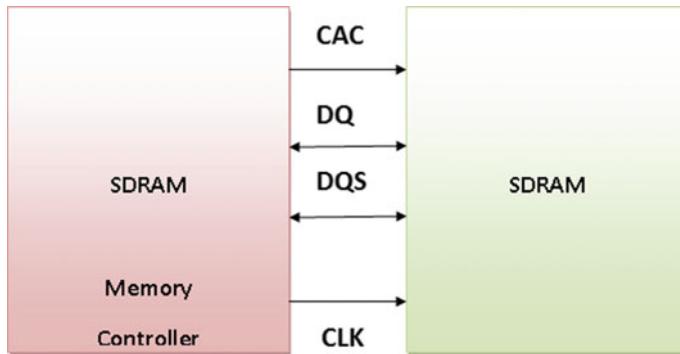


Fig. 7.12 SDRAM interface signals

Table 7.2 SDRAM interface signal description

Interface_signal	Description
CAC	Command address and control bus
DQ	Bidirectional data bus
DQS	Bidirectional data strobe
CLK	Clock for the SDRAM interface

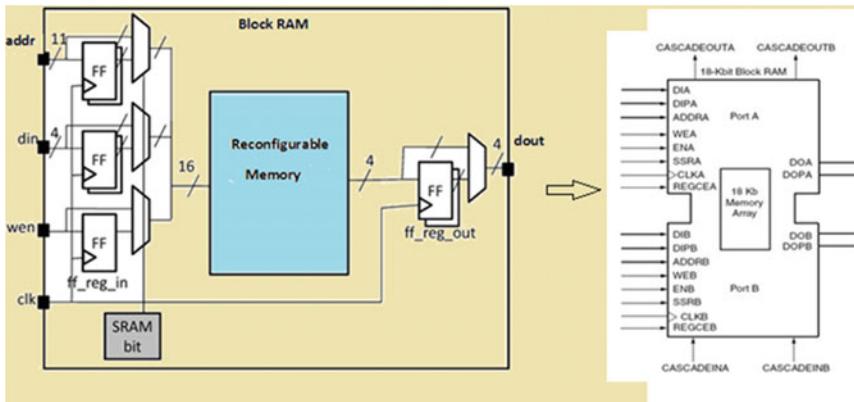


Fig. 7.13 BRAM structure [2]

BRAMs are used for the internal storage of the data, to design FIFO, buffers, stacks, and can be used to store data for the FSMs.

Every BRAM has the clock and clock enable, read, write, and every BRAM can be configured as synchronous RAM. If we consider the two-port BRAM, then both ports can be interchangeably used and can be controlled for the synchronous read/write operation. If we consider Spartan 3 devices, then it has BRAM which works at

```

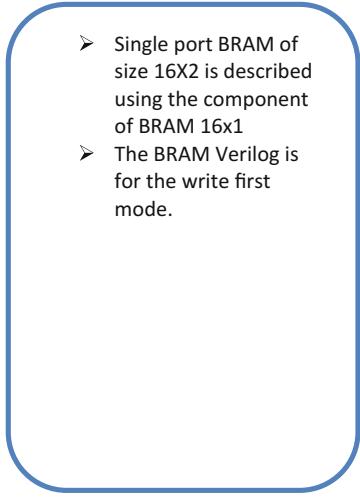
module BRAM_16to2 (clk, write_en , enable , addr_in , data_in , q_out);

input clk;
input write_en;
input enable;
input [3:0] addr_in;
input [1:0] data_in;
output wire [1:0] q_out;

reg [1:0] BRAM_mem [0:15];
reg [3:0] read_address ;

always @ (posedge clk)
begin
if (enable)
if (write_en)
begin
    BRAM_mem[addr_in] <= data_in;
    read_address <= addr_in;
end
end
assign q_out = BRAM_mem[read_address];
endmodule

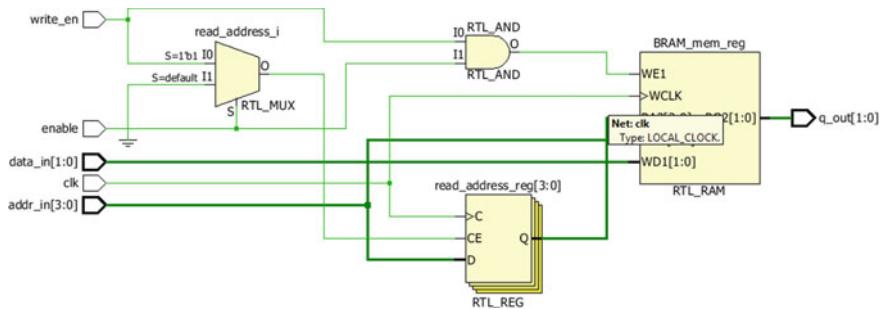
```

- 
- Single port BRAM of size 16X2 is described using the component of BRAM 16x1
  - The BRAM Verilog is for the write first mode.

**Example 7.6** Synthesizable Verilog RTL using BRAM component

200 MHz operating frequency. The BRAM single-port and dual-port structure is shown in Fig. 7.13.

As shown in Fig. 7.13, the BRAM consists of the reconfigurable memory, address lines, write enable and clk, data input and data output lines. The Verilog RTL for the inference of the BRAM is described in Example 7.6, and the synthesis result for the  $16 \times 2$  BRAM is shown in Figs. 7.14 and 7.15.



**Fig. 7.14** Synthesis result of BRAM [2]

**Fig. 7.15** Device utilization summary for Virtex-7

Resource	Estimation	Available	Utilization %
FF	4	728400	0.01
LUT	5	364200	0.01
Memory LUT	4	111000	0.01
I/O	11	600	1.83
BUFG	1	32	3.12

## 7.6 Memory Controllers

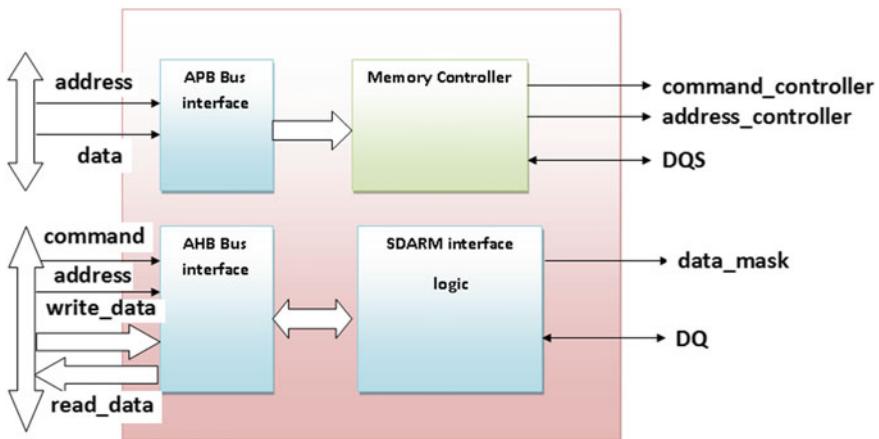
As discussed in the previous section, to access the data from the external memory we need to have the memory controllers. During this decade, most of the modern FPGA consists of the soft and hard memory controller cores. What the prototype team need to do is to understand the

1. External interfaces of the memory controller
2. Timing and latency (useful for the constraints)
3. Compatibility of the interface signals with the prototyping environment
4. Overall speed of the design
5. Are the soft cores compatible with the FPGA logic (or need to tweak the interfaces)
6. Is there mechanism to calibrate for the PVT variations?
7. Whether the interface supports LVDS standard or not?

Figure 7.16 shows the external interface signals generated from the memory controller and interface logic. With the processing environment, the communication can be established using the AHB and APB bus.

The command controller can be used to generate the following commands.

- Refresh
- Read
- Write
- Precharge
- Activate



**Fig. 7.16** DDR memory controller and AHB–APB interface

- Mode register set
- Extended mode register set

## 7.7 How This Discussion Is Helpful in SOC Prototyping?

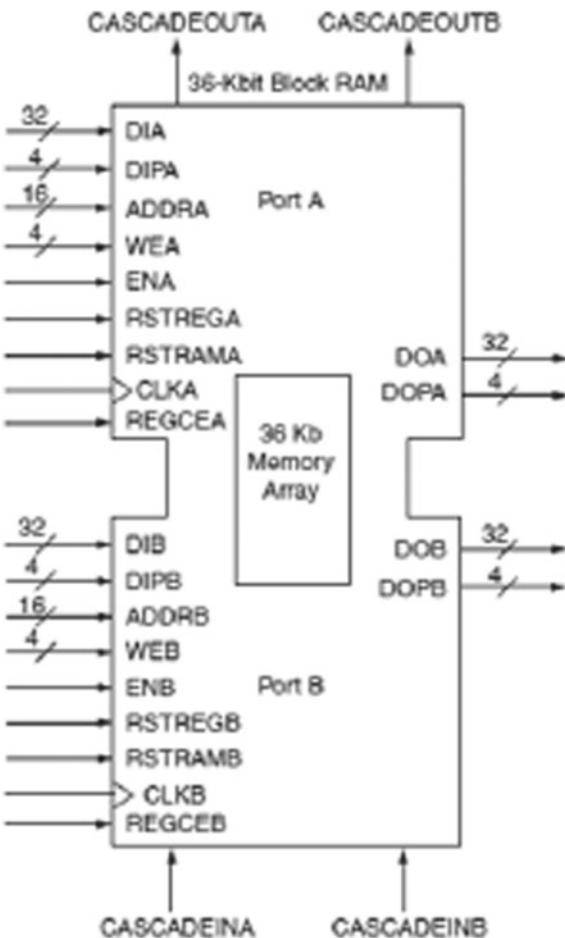
During prototyping, we can use the available soft and hard processor cores for the memory controller. If the requirement is to have the internal memory, then use the BRAMs. If large memory is required, then use the memory controller cores. The Intel Stratix 10 memory controller core is described in this section.

### 7.7.1 Xilinx 7 Series Block RAM

The BRAMs are used in many applications and the architecture of BRAM is vendor dependent. They can be configured using vendor-dependent EDA tool for the required capacity. The Xilinx 7 series architecture has 36 KB BRAM, which can be visualized as 2 X 18 KB BRAM. The BRAM is synchronous RAM and can be cascaded without any logic overheads to get 64 K X 1. The BRAM can be used as single port and dual port. In the dual-port mode, the 18 KB BRAM can be used and configured as 18 K X 1, 9 K X 2, 8 K X 4, 4 K X 9, etc., and 36KB BRAM can be used as 1 K X 36, 2 K X 9, 4 K X 9, etc. The BRAM architecture has built-in error correction (64-bit ECC), and they can be used also in FIFO mode (Fig. 7.17).

All kinds of the SOC design uses the memories of type RAM, ROM, content addressable. So, let us think that how these kinds of memories can be implemented.

**Fig. 7.17** Xilinx 7 Series  
BRAM [1]



The memories can be instantiated either from the cell library or from the memory generator.

To implement the small memories of a few bits, the LUTs can be used. It is important that these memories can be efficient enough to load, store, and pass the data. But to have the better and efficient architecture for the design instead of distributing the memories over the FPGA fabric, it is always better to use the BRAMs. The main features of BRAM are:

1. **Synchronous Memory:** BRAMs can implement the synchronous single or dual-port memory. One of the real beauties of such memory blocks is that when configured as dual-port RAM each port can operate at different clock frequencies.
2. **They can be configured:** The BRAM block is dedicated dual-port synchronous RAM block and can be configured as discussed above. Each port can be configured independently.

3. **BRAMs and their use in the FIFO designs:** The BRAMs can be used to store the data as they are dedicated and configured. With additional logic, FIFOs can be implemented using BRAMs. The depth of the FIFOs can be configured with the restriction that both the read and the write sides should have same width.
4. **Error Correction:** Consider the BRAM is configured as the 64-bit RAM, and then each BRAM can store additional Hamming code bits. These bits are used to perform the single-bit and double-bit error corrections during the read process. For 64-bit BRAM, each BRAM can store the eight-bit Hamming code. The error correction logic can be also used while writing or reading from the external memories.

### So, let us think how the BRAMs are inferred?

The synthesis tool partitions the larger memories into small blocks, and each block can be implemented using BRAM. Effectively in the simple words, the BRAMs are very effective building block which is inferred automatically during synthesis and they are combined to model the wide range of memories used for the SOC.

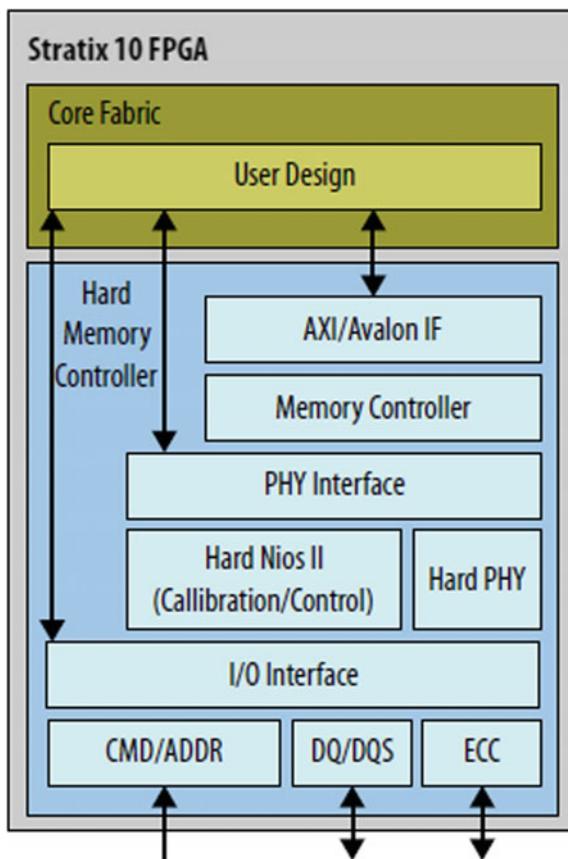
## 7.7.2 *Stratix 10 Memory Controllers*

1. Intel Stratix 10 devices offer external memory bandwidth, with up to ten 72-bit wide DDR4 memory interfaces running at up to 2666 Mbps.
2. The memory controllers have the lower power and resource efficiencies of hardened high performance.
3. The external memory interfaces can be configured up to a maximum width of 144 bits when using either hard or soft memory controllers (Fig. 7.18).

### 7.7.2.1 Memory Controller's Important Features

1. **Memory controllers in IO bank:** Each I/O bank contains 48 general purpose I/Os and a high-efficiency hard memory controller. Controller is capable of supporting many different memory types, having the different performance capabilities.
2. **Hard and soft memory controllers:** The hard memory controller is also capable of being bypassed and replaced by a soft controller implemented in the user logic. The I/O has a hardened double data rate (DDR) read/write path (PHY) capable of performing key memory interface functionality
  - Read/write levelling
  - FIFO buffering to lower latency and improve margin
  - Timing calibration
  - On-chip termination

**Fig. 7.18** Stratix 10 memory controller [2]



3. **Multiple memory interface calibrations:** The timing calibration is included using the hard microcontrollers based on Intel's Nios® II technology, specifically used to control the calibration of multiple memory interfaces. This calibration allows the Intel Stratix 10 device to compensate for any changes in process, voltage, or temperature either within the Intel Stratix 10 device itself or within the external memory device.
4. **Advanced calibration:** The advanced calibration algorithms ensure maximum bandwidth and robust timing margin across all operating conditions (Table 7.3).
5. **Parallel and serial memory interface:** In addition to parallel memory interfaces, Intel Stratix 10 devices support serial memory technologies such as the Hybrid Memory Cube (HMC). The HMC is supported by the Intel Stratix 10 high-speed serial transceivers, which connect up to four HMC links, with each link running at data rates of 15 Gbps (HMC short reach specification).

**Table 7.3** Stratix 10 hard and soft memory controller performance [2]

Interface	Controller type	Performance (Mbps)
DDR4	Hard	2666
DDR3	Hard	2133
QDR II/II+Xtreme	Soft	550
RLDRAM II	Soft	533
RLDRAM II	Soft	2400

6. **LVDS IOs:** Intel Stratix 10 devices also feature general purpose I/Os capable of supporting a wide range of single-ended and differential I/O interfaces. LVDS rates up to 1.6 Gbps are supported, with each pair of pins having both a differential driver and a differential input buffer. This enables configurable direction for each LVDS pair.

## 7.8 Important Takeaways and Further Discussions

Following are the few of the important points to conclude this chapter

1. The SRAM does not need the refresh circuit. DRAM needs the refresh circuit.
2. In the double data rate memories; the data can be transferred on the positive and negative edge of the clock.
3. The data alignment logic needs to be incorporated for the DDR. The data can be aligned at the center of every clock edge.
4. The latency and timing of the address select and data select decide the speed of the data transfer.
5. In the FPGA, the memories can be implemented using the LUTs or ALM blocks.
6. For large memory blocks, the BRAMs can be the better choice.
7. Xilinx Virtex-7 and Stratix 10 devices have the in-built hard memory controller cores.

## References

1. [www.xilinx.com](http://www.xilinx.com)
2. [www.altera.com](http://www.altera.com)

# Chapter 8

## DSP Algorithms and Video Processing



*The DSP processing environment should use the real-time clocks, DSP processor cores, and DMA.*

**Abstract** This chapter discusses the DSP algorithms and the role of the design engineer to achieve the desired performance for the DSP designs. The chapter is useful to understand the basics of FIR, IIR filter design using Verilog and the performance improvement for the design. The chapter even focuses on the architecture and micro-architecture and their implementation for the video applications. The video encoder and decoder architectures and micro-architecture to design them are discussed with the practical scenarios.

**Keywords** RTL · Verilog · DSP · FIR · IIR · LFSR · Video decoder · Video encoder · Audio processing · Video processing · If-else · Case · Process · Sequential design · Pipelining · DSP processor · MAC

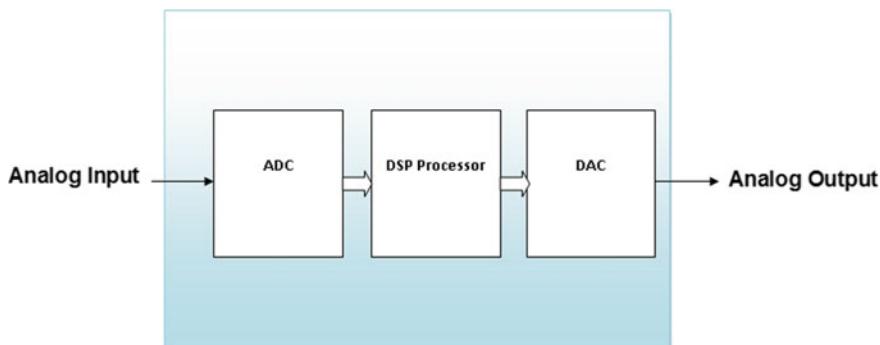
During this century, we visualize lot many applications using the digital signal processing (DSP). The complexity of these applications and the desired speed encourages us to design the high-performance DSP processors. The application can be in the multimedia, audio or video processing; the requirement is to have the least area, low power and high speed. Even the data rate, efficiency of the design, and multitasking are key important parameters need to be thought before implementing such applications. The chapter focuses on all these aspects and the design of the DSP algorithms. The chapter is useful to understand the DSP processor trends, architecture, and the Xilinx and Intel FPGAs suitable for such kind of the complex applications.

## 8.1 DSP Processors

If we consider the use of the DSP, then there are many areas in which we can use the algorithms to get the desired speed, power, and area. The main applications of the DSP are in the following areas, these areas are evolved during this decade, and still there is research going on in these areas. Few of the DSP application areas are listed below:

1. **Control and instrumentation applications:** The DSP processors and algorithms are extensively used for navigation and guidance, power system monitoring, transient analysis of the signals, RADAR, sonar, etc.
2. **Speech processing applications:** Most of the times, we need to have the efficient DSP algorithm or architecture for the encryption, decryption, and speech recognition. For such kind of designs, the DSP algorithms can be realized using the FPGAs.
3. **General Purpose DSP applications:** In general, most of the time to design the filters like FIR, IIR and for convolution we use the DSP algorithms using C/C++ or using the HDLs.
4. **Audio processing applications:** Audio equalization, audio mixing, sound synthesis are few of the important applications where the efficient DSP architecture and implementation can give the better results.
5. **Image processing:** The compression and decompression of the images, image recognition, face recognition and image enhancement are few of the areas where DSP algorithms and processors are used extensively.

In this context, the designs need to be prototyped on the FPGAs. In the present scenario, if we consider the modern FPGA architecture of Xilinx or Intel FPGA, then we can conclude that these architectures are efficient enough to achieve the desired performance for the complex DSP tasks. If we consider the basic DSP representation, then we can think of following blocks to implement the DSP algorithms (Fig. 8.1).



**Fig. 8.1** DSP-based processing blocks

As shown in the figure for any DSP implementation, we can think of using the following blocks:

1. ADC
2. DSP processor
3. DAC

The input to the system is analog input and is converted into the digital data using the ADC. The analog inputs are sampled by the sample and hold depending on the sampling frequency and resolution of the ADC. The sampled signals are quantized to get the digital output and given to the DSP processor for the processing of the desired application. The output from the system can be digital or analog depending on the requirement of the design. In the practical environment the system may need to have input filters for such kind of processing.

As a prototype engineer to implement these applications, we need to think about the following few points:

1. How fast the input signal is? This can allow us to choose the ADC to sample the correct signal.
2. In the practical scenario, the designer can use the ADC daughter card with the FPGA.
3. The DSP algorithm complexity, speed, power, and bandwidth requirements decides about the selection of the desired FPGA.
4. Whether the design needs hard processor, DSP core or the DSP algorithms need to be implemented using HDL?
5. What is the operating frequency required to execute the single instruction or multiple instructions?
6. Whether my design architecture is efficient enough to allow the chunk of data residing inside the FPGA platform?
7. Can I use the lower frequency with the multitasking or whether my design needs to run at high frequency without parallelism?
8. Whether it supports the real-time processing of the data?

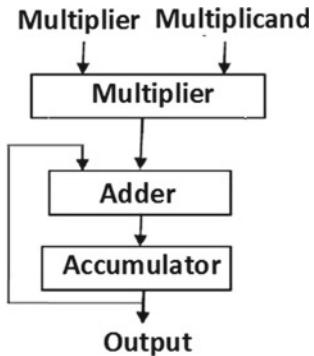
The answer to all these questions can yield into the better DSP architecture and the algorithm implementation.

## 8.2 DSP Algorithms and Implementation

### What we need to think is?

1. What kind of computational elements required?
2. The complexity of the DSP algorithm
3. What are the functional implementation requirements
  - a. Adders, multipliers, shifters, MAC
  - b. The pipelined requirements

**Fig. 8.2** Multiply and accumulate



- c. The speed, area, and low power requirements for the design
- d. The design partitioning into multiple functional blocks

For example, consider the DSP algorithm which needs accumulation of the data after multiplication. The MAC can be efficiently designed and shown in Fig. 8.2.

The RTL design and implementation of the Linear Feedback Shift Register (LFSR) are discussed in this section. The designers can think of other algorithmic implementation like FIR, IIR using the HDLs.

### 8.2.1 LFSR

In most of the applications, we need to implement the polynomial to have the LFSR. The RTL code for the polynomial is described in Example 8.1 (Fig. 8.3).

## 8.3 DSP Processing Environment

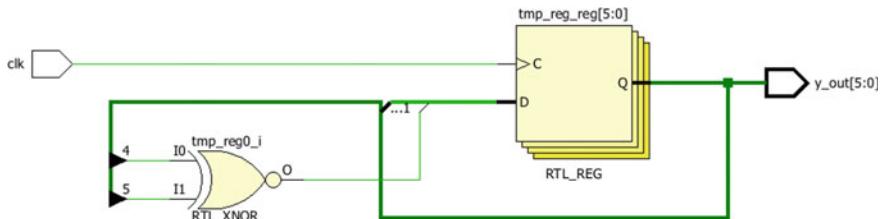
While designing algorithms for the DSP applications, consider the following few important points (Fig. 8.4).

1. The processing speed, throughput, and the IO data rate
2. The processor architecture, whether it supports the pipelining of the instructions
3. Whether processor supports the floating point operations using the available instructions
4. The architecture should have the separate program memory and data memory buses
5. For fast access of the data, the DMA interface should be the better choice.
6. Have the internal storage in the form of FIFO or circular buffers

```
//verilog code for the lfsr
module lfsr ( clk, y_out);
    input clk;
    output [5:0] y_out;
    reg [5:0] tmp_reg;
    integer k;
    always @ (posedge clk)
    begin
        tmp_reg [0] <= tmp_reg[4] ~^ tmp_reg[5];
        for (k=5; k>=1; k=k-1)
            tmp_reg [k] <= tmp_reg [k-1];
    end
    assign y_out = tmp_reg;
endmodule
```

➤ The LFSR triggered on the positive edge of clock and having output  $y_{out}$ .

**Example 8.1** Verilog code for the LFSR



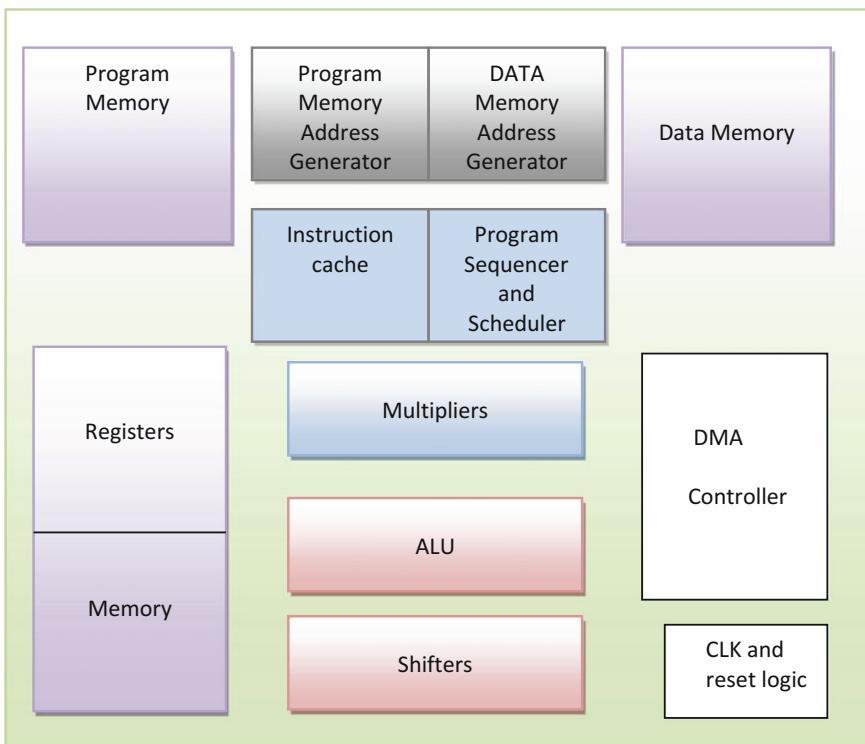
**Fig. 8.3** Synthesis result for the LFSR

## 8.4 Architecture for the DSP Algorithms

While architecting for the SOC for the DSPs what we need to think? This can be effectively answered in the following way!

1. **DSP processor core:** The capability of core should be to perform the complex operations. It should have

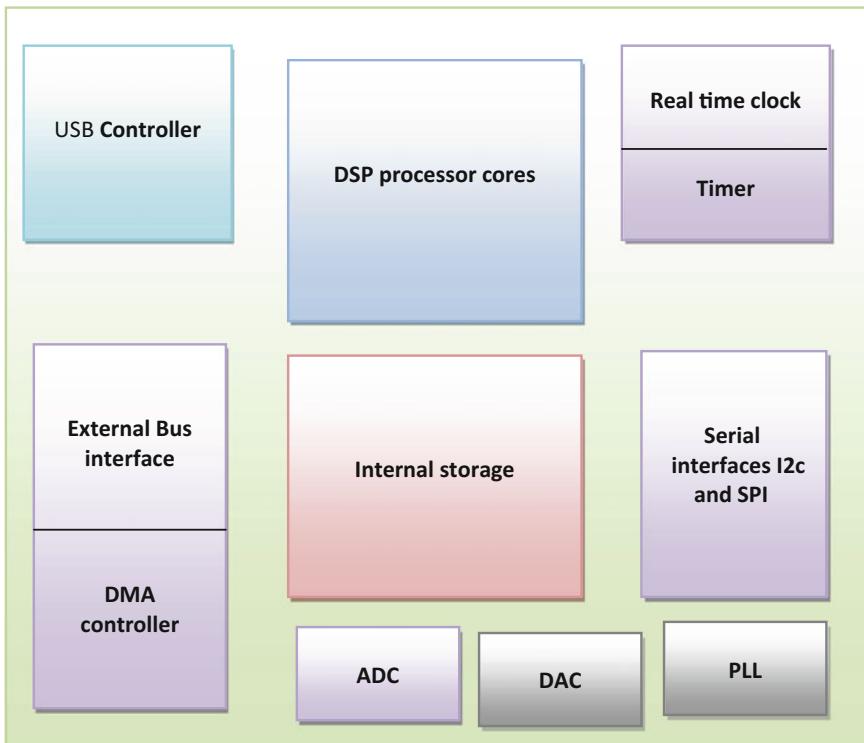
- a. Internal memory and storage registers
  - b. Circular buffers and FIFO mechanisms to support the queuing of the data
  - c. Multipliers and large-density accumulators
  - d. Shifters
  - e. The floating point support logic
  - f. The separate logic for the program and data memory access
  - g. Pipelining and multitasking features
2. **DMA controllers:** The most important feature of the direct memory access should be on chip with the processor. This will give freedom to the DSP processors to perform the concurrent operation with the DMA.
- a. DMA can be used to transfer the burst of the data between the memories or from the memory to IO.
3. **The serial interfaces:** The capabilities like serial data transfer using I2C or SPI can be an added advantage. They can be used to interface the external serial devices with the SOC.



**Fig. 8.4** DSP processing system

4. **On-chip PLL:** The phase locked loops to generate the clock and the additional clock distribution logic for the clock distribution with uniform clock skew.
5. **Real-time data processing:** To process the real-time data, the timers and real-time clocking should be present in the DSP SOC.
6. **USB controllers:** The USB interface for the data transfer between the host system and the DSP processor core can be used in most of the architectures.
7. **Analog blocks:** The analog blocks like ADCs and DACs can make the SOC compatible for the analog interfaces.
8. **BUS interface logic:** The SOC components with additional logic can be interfaced with the host using the high speed bus interfaces.

By considering all these features, the architecture of the SOC should be evolved for the required DSP capabilities is shown in Fig. 8.5.



**Fig. 8.5** Architecture for the DSP processing system

## 8.5 Video Encoders and Decoders

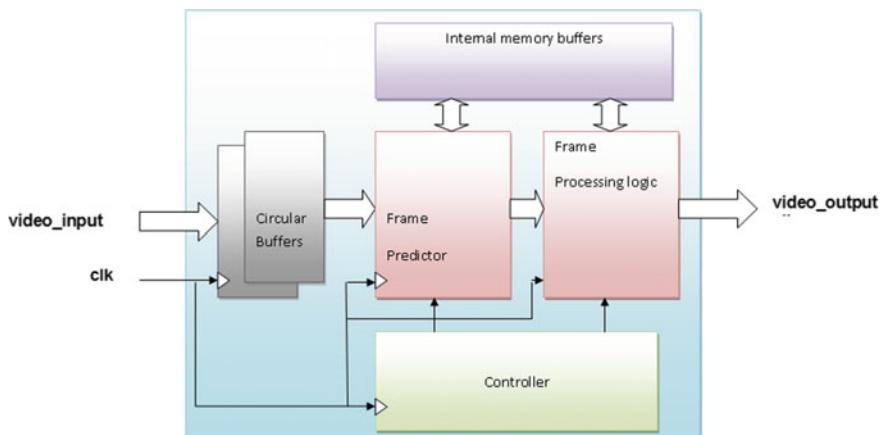
The high-density video processing systems need to have the video encoders and the decoders. The architecture of such type of the system is complex enough due to the parallelism, storage needs. The video encoding should be real time and what needs to be incorporated?

1. The ping-pong buffers or the circular buffers to queue the data.
2. Frame prediction logic can be used to detect the type of the frame. For example, if we use the H.264 encoder standard, then the frame can be intra or inter and can be predicted by the frame prediction logic.
3. Frame processing logic: To have the quantization and the entropy coding, the logic can be used. As such kind of system uses the complex matrix multiplications, the density of such logic is high.
4. Internal memory buffers: To store the data for the predictions, the high capacity memory buffers are required.
5. Controllers: The controller using the multiple state machines can be designed for such kind of the encoders to derive the timing and control signals.

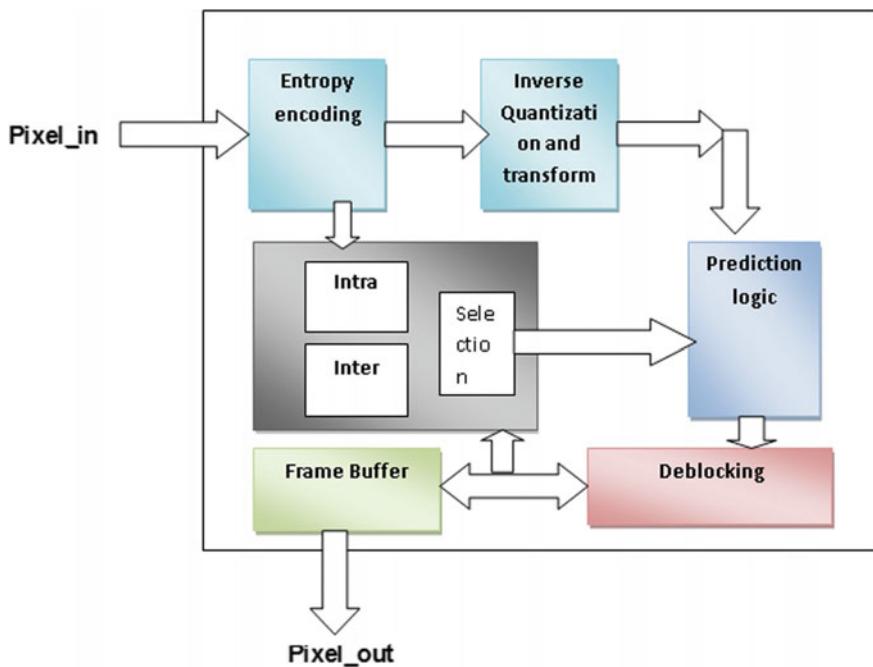
The video encoding system is shown in Fig. 8.6. It is assumed that the video input and output are digital data.

The compressed video data from the video encoders can act as input to the video decoding system (Fig. 8.7). The components of such type of system are

1. Entropy encoding
2. The intra- or inter frame prediction logic
3. Inverse quantization and transform
4. Deblocking logic
5. Frame buffer (frame storage)



**Fig. 8.6** Video encoder



**Fig. 8.7** Video decoder (H.264)

## 8.6 How the Discussion Is Helpful in SOC Prototyping?

As the FPGAs are having the DSP capabilities, they can be used to implement the DSP algorithms. During the SOC prototyping, the RTL can be tweaked to have the FPGA equivalent. Implement the FPGA-based algorithms by using the dedicated DSP blocks available inside the FPGA.

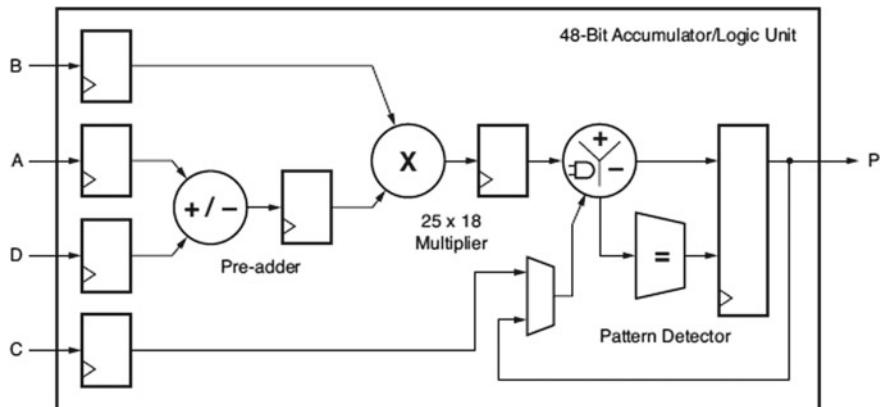
High-density FPGAs from Xilinx or Intel are efficient for digital signal processing (DSP) applications because they can implement custom, fully parallel algorithms. As stated earlier, the DSP system should use the multipliers and accumulator while executing the DSP algorithms.

Features of Xilinx 7 series FPGAs are listed below:

1. Full-custom, low-power DSP slices
2. High-speed, small size architecture
3. The DSP slices to enhance the design performance.

The basic functionality of the DSP48E1 slice is shown in Fig. 8.8 [1], and highlights of the DSP functionality include [1]:

1.  $25 \times 18$  two's complement multiplier
2. Dynamic bypass 48-bit accumulator
3. Single instruction, multiple data (SIMD) arithmetic unit



**Fig. 8.8** Xilinx DSP slice architecture [1]

4. Dual 24-bit or quad 12-bit add/subtract/accumulate
5. 96-bit-wide logic functions when used in conjunction with the logic unit
6. Optional pipelining and dedicated buses for cascading.

### 8.6.1 Intel FPGA DSP Block

Intel Stratix 10 devices have the powerful DSP features to perform the floating point operations. The DSP block has the hard fixed point capability. The DSP architecture is based on the variable precision architecture.

The important features of the DSP block are listed below:

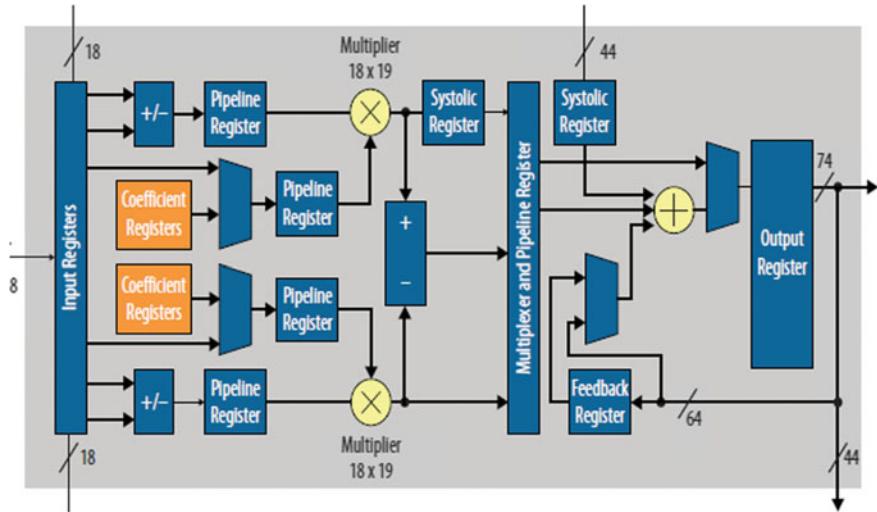
1. Hard 18-bit and 25-bit pre-adders
2. Hard floating point adders and multipliers
3. For separate  $I, Q$  product accumulation the provision of the 64-bit accumulator
4. Embedded coefficient registers for the 18-bit and 27-bit coefficients
5. Cascaded output adder chain for 18- and 27-bit FIR filter
6. Fully independent multiplier output
7. Can be easily inferred in all the modes using the HDL

The DSP block having standard precision fixed point mode is shown in Fig. 8.9 [2].

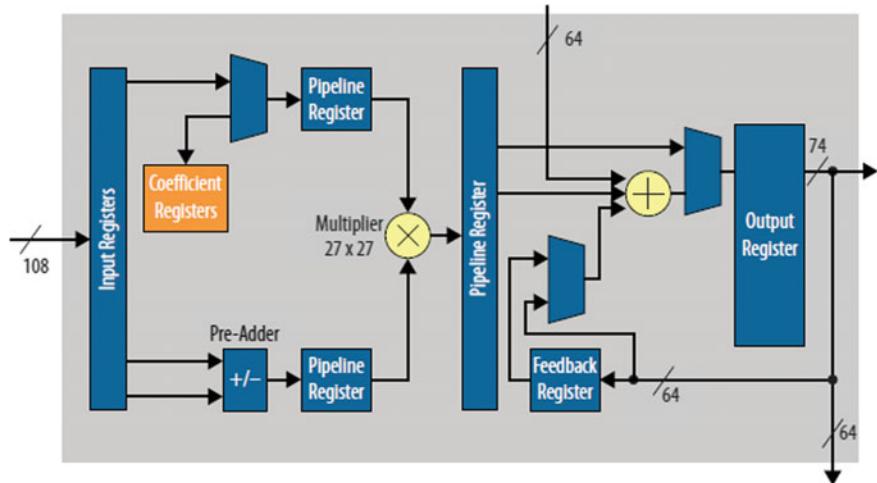
The DSP block with high precision fixed point mode is shown in Fig. 8.10 [2].

The DSP block with the single precision floating point number is shown in Fig. 8.11 [2].

As shown in the figure, each DSP block can be configured independently as either dual  $18 \times 19$  or single  $27 \times 27$  multiply and accumulate. The main application of such kind of DSP is to implement the high precision DSP functions using the 64-bit

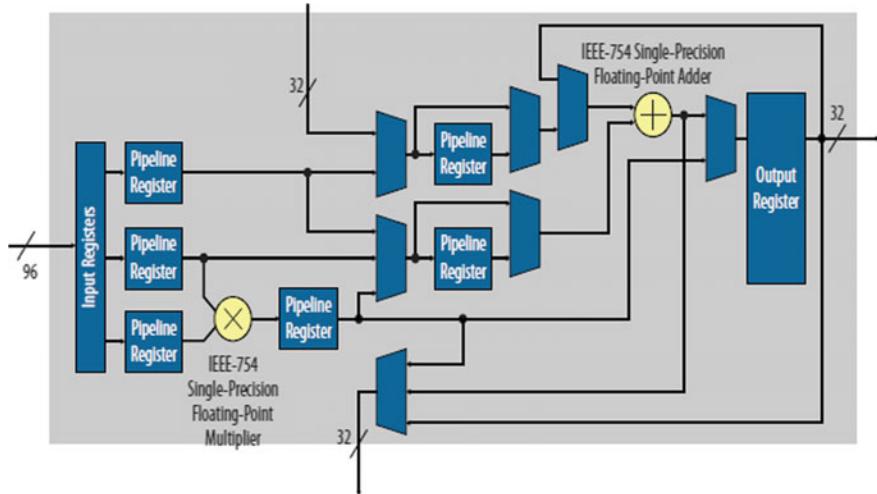


**Fig. 8.9** Standard precision fixed point mode [2]



**Fig. 8.10** High precision fixed point mode [2]

cascade bus. The architecture of the DSP block is flexible enough, and by using 64-bit bus the multiple high precision blocks can be cascaded. Even in the floating point mode, each DSP block provides the single precision floating point adder and multiplier.



**Fig. 8.11** Single precision floating point number [2]

Table 8.1, shows the variable precision DSP block configuration.

The complex multiplication using variable precision DSP block supports the FFT algorithms. The DSP block supports 18-bit DSP applications such as high-definition video processing. It supports the floating point multiplications. The major advantage of using this DSP capability is to reduce the overheads in the system design. It has increased system performance and the low power consumption.

The prototype team can choose the FPGA depending on the need of the DSP capabilities and complexity.

## 8.7 Design Scenarios

This section describes a few of the design scenarios. Most of the time, we need to have the multipliers, barrel shifters, and filters during the implementation of the DSP algorithms.

### 8.7.1 *The Design of the IIR Filter*

The infinite input response filter implementation is described in Example 8.2.

**Table 8.1** Multiplier size [2]

Multiplication size	DSP block resources	Expected usage
18 × 19 bits	Half of variable precision DSP block	Medium precision fixed point operation
27 × 27 bits	One variable precision DSP block	High precision fixed point operation
19 × 36 bits	One variable precision DSP block with external adder	Fixed point FFT
36 × 36 bits	Two variable precision DSP blocks with external adder	Very high precision fixed point
54 × 54 bits	Four variable precision DSP blocks with external adder	Double precision floating point
Single precision floating point	One single precision floating point adder, one single precision floating point multiplier	Floating point

### 8.7.2 FIR Filter

The Verilog description of the direct FIR filter is described in Example 8.3. The filter design uses more number of multipliers. While implementing the filters, the designer can push the logic using the DSP slices (Figs. 8.12 and 8.13).

### 8.7.3 Barrel Shifters

The barrel shifters are used to shift the data during the DSP algorithms. The Verilog code is described in Example 8.4 (Fig. 8.14).

```
//Verilog code for the iir filter
module iir_design (clk, reset_n, data_in, data_out);
parameter N=15;
input clk ;
input reset_n;
input [N-1:0] data_in;
output [N-1:0] data_out;
reg [N-1:0] tmp1_data_out, tmp2_data_out;
always@ (posedge clk or negedge reset_n)
begin
if (~reset_n)
tmp1_data_out<=0;
tmp2_data_out<=0;
else
tmp1_data_out<=data_in;
tmp2_data_out<= tmp1_data_out+
{tmp2_data_out[N-1], tmp2_data_out[N-2:0]}
+ {2{tmp2_data_out[N-1], tmp2_data_out[N-1:1]}};
end
assign data_out<= tmp2_data_out;
endmodule
```

➤ The IIR filter sensitive on the rising edge of clock.

**Example 8.2** Synthesizable Verilog code of the IIR filter

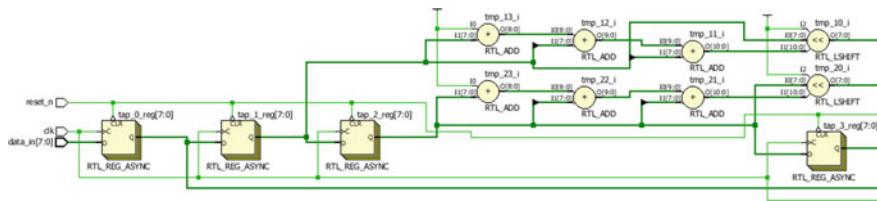
```
//Verilog code for the 4 tap direct FIR filter
module fir_design (clk, reset_n, data_in, data_out);
parameter N=8;
input clk ;
input reset_n ;
input [N-1:0] data_in ;
output [N-1:0] data_out ;
reg [N-1:0] tmp_0, tmp_1, tmp_2, tmp_3;
reg [N-1:0] data_out, tap_0, tap_1, tap_2, tap_3;

always @ (posedge clk or negedge reset_n)
begin
if(~reset_n)
begin
data_out<=0;
{tmp_0,tmp_1,tmp_2,tmp_3} <= {0,0,0,0};
tap_3<=0;
tap_2<=0;
tap_1<=0;
tap_0<=0;
end
else
begin
tmp_1 <= tap_1<<1+tap_1+{tap_1[7], tap_1[7:1]}+{tap_1[7], tap_1[7], tap_1[7:2]};
tmp_2 <= tap_2<<1+tap_2+{tap_2[7], tap_2[7:1]}+{tap_2[7], tap_2[7], tap_2[7:2]};
tmp_3<=tap_3;
tmp_0<=tap_0;
data_out<= tmp_1+tmp_2-(tmp_3+tmp_0);
tap_3<=tap_2;
tap_2<=tap_1;
tap_1<=tap_0;
tap_0<=data_in;
end
end
endmodule
```

➤ The four tap direct fir filter realization using the non-blocking assignments

➤ The four tap direct FIR filter using Verilog is sensitive to positive edge of the clock and having the output data\_out

**Example 8.3** Synthesizable Verilog code of four tap FIR filters



**Fig. 8.12** Synthesis result for the FIR filter

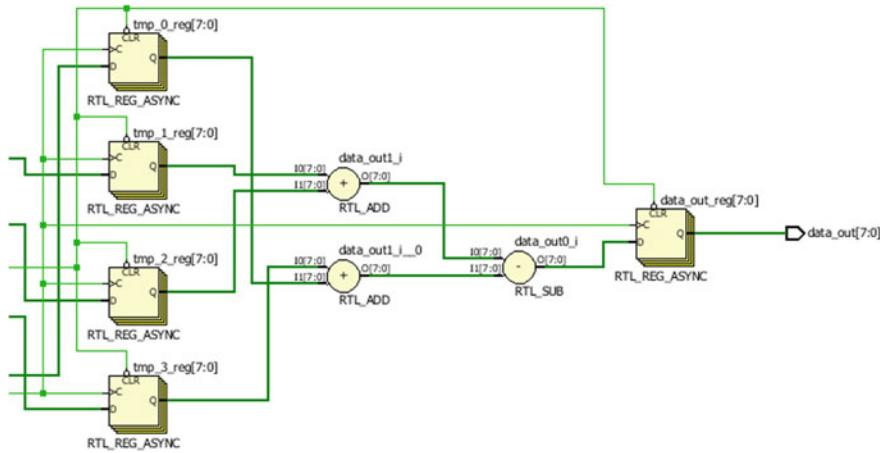
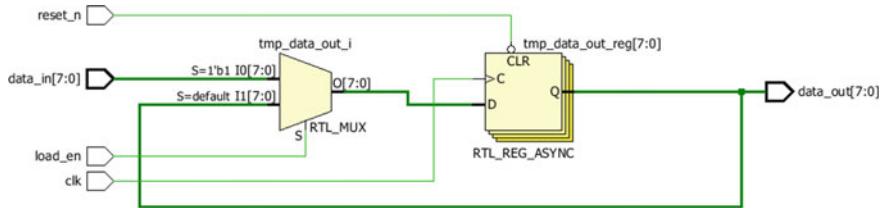
```
// Verilog code for the barrel shifter
module barrel_shifter (clk, reset_n, load_en, data_in, data_out);

input clk;
input reset_n ;
input load_en;
input [7:0] data_in ;
output wire [7:0] data_out ;
reg [7:0] tmp_data_out;

always@ (posedge clk or negedge reset_n)
begin
if(~reset_n)
tmp_data_out<= 0;
else if (load_en)
tmp_data_out<= data_in;
else
tmp_data_out <= {tmp_data_out[6:0],tmp_data_out[7]};
end
assign data_out = tmp_data_out ;
endmodule
```

➤ The data is shifted on the rising edge of the clock

**Example 8.4** Synthesizable Verilog code for the barrel shifter

**Fig. 8.13** Synthesis result for the FIR filter(contd.)**Fig. 8.14** Synthesis result for the barrel shifter

## 8.8 Important Takeaways and Further Discussions

Following are a few important points to summarize the chapter

1. DSP applications like IIR, FIR need to be implemented using the dedicated FPGA blocks.
2. If DSP IPs are available, then during prototype use the vendor specified boards.
3. If the multiple FPGAs are used in the design, then take care of the design partitioning of complex IIR and FIR filters.
4. Choose the DSP processors by understanding the required speed requirements for the design.
5. Use the pipelined algorithms and pipelined controlled stages to implement the DSP algorithms.
6. For floating point operations, the important parameters are area, speed, and power. If FPGAs are used, then check for the inference of the RTL code in the specific DSP slice.

The next chapter discusses the ASIC and FPGA synthesis and is useful to have an understanding of the synthesis and constraints. The RTL tweaks required for the FPGA equivalent are discussed in the chapter.

## References

1. [www.xilinx.com](http://www.xilinx.com)
2. [www.altera.com](http://www.altera.com)

# Chapter 9

## ASIC and FPGA Synthesis



*ASIC and FPGA syntheses differ in many aspects. During SOC prototyping tweak the RTL to have the FPGA equivalent logic inference.*

**Abstract** The chapter discusses the synthesis for the ASIC and FPGA. During the ASIC prototyping, FPGAs are used and how the ASIC designs can be migrated to FPGA which is discussed in this chapter. The chapter focuses on the important RTL design concepts design portioning, block-level and chip-level synthesis to start with. The design constraints used during the synthesis are discussed in this chapter with the practical scenarios. The chapter also focuses on the Synopsys DC commands used during synthesis. The gated clocks and implementation for the ASIC and FPGA are discussed with the implementation scenarios.

**Keywords** Synthesis · ASIC · FPGA · Block-level synthesis  
Chip-level synthesis · Constraints · Area · Speed · Power · Clock gating  
Partitioning · Combinational loops · Latch inference · LUTs · CLB · Slice  
Standard cells · Macros · Hard macros · Soft macros · IPs · FSM · Optimization  
Logic duplication

### 9.1 Design Partitioning

The design partitioning at the top level for the better optimization and clean timing paths can play important role for any ASIC/SOC designs. The thought should come in the mind of the architect to achieve the better hardware and software partitioning. Consider a design scenario in which billion gate SOC has the processors, analog IPs or blocks and other digital logic and the configuration is controlled by the software commands. Under such circumstances, it is better to have following clarity while partitioning the design.

1. Can I partition the analog and digital blocks and isolate the power domains? Answer is yes, and it is requirement of the design.
2. Can I have the partition for the digital logic depending on the multiple power domains? Yes that is required to achieve the low power. Always partition the design depending on the voltage domains so that functional blocks can be power down or power up as and when required.
3. The large density digital functional blocks can be partitioned into moderate gate count blocks and care should be taken to have the registered inputs and registered outputs at the interface level of the blocks.
4. Partition the design depending on the clock domains. Let us have the separate RTL design for the individual functional blocks sensitive on the different clock edges. At the top level during the RTL integration stage, use the synchronizers to pass the data between multiple clock domains.
5. If the design needs to be realized using the FPGA, then partition the design as digital and analog equivalent. Use the single or multiple FPGAs to realize the digital design. Use the external boards with the required interfaces to implement the analog blocks such as ADCs, DACs, clocking structures and power supply and temperature monitoring/cooling blocks.
  - a. In the multiple FPGA design, check for the IO availability and interface timing while partitioning the design and it will be discussed in more details in Chaps. 13–15.
  - b. Use the high-speed IOs to transfer the data between the FPGA.
  - c. Use the required bus topology as daisy chaining, star or mixed interconnect.
6. The larger FSM controller's logic should be partitioned to achieve the better area, speed, and timing. Use the gray or one-hot encoding method to realize the logic.
7. Have the information about the latency, data rate, and desired interface timing while partitioning the design.
8. For the hardware and software partitioning, consider the performance requirement and use the FIFO or buffers to queue the chunk of data.
9. Have the implementation of the algorithm such as high-speed buses, processors, DSP blocks, video decoders using RTL and use the software for configuring the logic at the initialization phase or to store larger amount of the data.

## 9.2 RTL Synthesis

For the complex SOC designs, the logic and physical synthesis step is important to achieve the desired performance for the design. During synthesis, the goal is to have the minimum area and power. The design should have high speed. It is essential to use the performance improvement techniques to achieve the better optimization for the area, speed, and power. The synthesis tool uses following to perform the synthesis

1. RTL design

2. The ASIC libraries
3. The design constraints

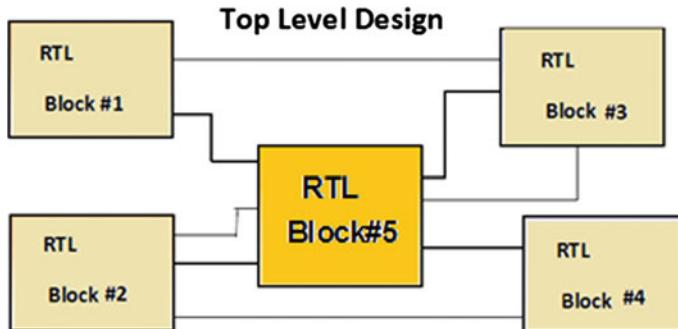
The outcome of the synthesis is the gate-level netlist. For the FPGA synthesis, the synthesis tool uses the target FPGA resources such as CLBs, IOBs, DSP blocks, BRAMs, hard processor IPs, RTL Design integration at top level is shown below (Fig. 9.1).

The Synopsys Design Compiler (SDC) reads the standard cell libraries and the RTL description to generate the technology-dependent gate-level netlist. Synthesis is the process of converting the design from the higher level of abstraction to the lower level of the design abstraction.

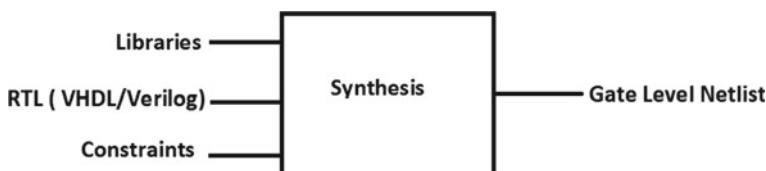
The synthesis tool should be efficient enough to optimize the design for the given constraints. The inputs to the synthesis tool are libraries, RTL description using either VHDL or Verilog, constraints. The output from the synthesis tool is gate-level netlist (Fig. 9.2).

The synthesis tool performs many steps to generate the required gate-level netlist, and these steps are described in Fig. 9.3.

As shown in the synthesis flow, the Design Compiler reads the DesignWare, technology, and symbol libraries to perform the synthesis. The synthesis tool is intelligent enough to identify the components from the DesignWare and technology library. The technology library consists of the logic gates, flip-flops, and latches. The complex components like comparators, adders, multipliers are part of the Design-

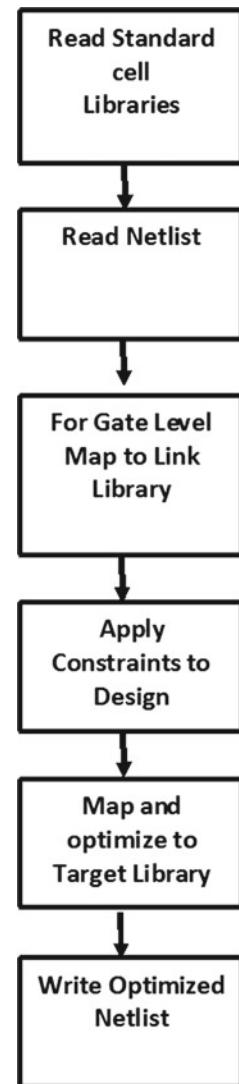


**Fig. 9.1** RTL design integration at top



**Fig. 9.2** Synthesis tool input and output

**Fig. 9.3** Synthesis steps for Synopsys DC



Ware library. By using the components efficiently either from the DesignWare or technology library, the DC performs the synthesis to generate the gate-level netlist.

In the next step, the DC reads the RTL either written in the VHDL or Verilog and proceeds to the next step where it can map to the link library if it is in the gate-level form. The synthesis tool performs the optimization of the RTL before mapping the cells to the technology or target library.

The synthesis tool uses the constraints, and those are area, speed, power, and even the environmental constraints while performing synthesis. The constraints are design specific and required for the desired or required outcome of the design.

The main important point is that the RTL designer and the synthesis team should know about the target standard cell library so that the design can be optimized to get the required functional outcome.

In the practical environment while using the Synopsys DC always care is taken about the tight constraints of area, speed, and power. There are many designs in which the restriction is there on the overall area utilization. For the million gate design, it is iterative process as it requires the intelligent design partitioning and even the tight constraints to get the desired performance. For complex ASIC designs, the team of RTL designer, synthesis, and timing teams works closely to realize the intended design functionality for the given constraints.

### 9.3 Design Constraints

The design constraints such as area, speed, and power need to be met, and this section discusses the few of the Synopsys DC commands used while constraining the ASIC designs.

Following steps are used by synthesis tool:

1. What synthesis tool does is that, it reads the DesignWare libraries, technology, and symbol libraries.
2. Second step is to read the RTL described by Verilog.
3. Perform the optimization to map the logic using the technology library, that is, also called as target library.
4. Use the design constraints like area, speed, and power and perform the optimization.
5. Map the design to target library and optimize the design.
6. Finally write the optimized netlist in the (.v) or (.ddc) format.

The sample design constraint script is given in Example 9.1.

Use the above script as clk.scr and to generate the reports use the following commands Useful Commands (9.1).

### 9.4 Synthesis and Constraints

Consider the top-level RTL design, which is integration of the different functional blocks. It is assumed that the design is partitioned in the better way to achieve the optimization for area, speed, and power. If we use the bottom-up approach for the design, then for every functional block, the block-level constraints should be met.

```

/*set the clock*/
set clock clk

/* set clock period */
set clock period 1

/* set the latency */
set latency 0.025

/* set clock skew*/
set early_clock_skew [expr $clock_period/10.0]
set late_clock_skew [expr $clock_period/20.0]

/* set clock transition */
set clock_transition [expr $clock_period/100.0]

/* set the external delay*/
set external_delay [expr $clock_period*0.2]

/* define clock uncertainty */
set clock_uncertainty -setup $early_clock_skew
set clock_uncertainty -hold $late_clock_skew

```

**Example 9.1** Basic clk.src script for 1 GHz design

**Useful Commands 9.1** Report generation commands

```

Dc_shell> report_timing
Dc_shell> report_clock
Dc_shell> report_constraints-all_violators

```

The block-level constraints to meet the desired area, speed, and the power are specified using the tcl'tk commands.

Why it is essential to specify the block-level constraints is one of the common questions? The few important points are listed below:

1. The complex SOC or ASIC designs can have the multiple functional blocks and may have multiple clock domains, power domains, and complex functionality.
2. For the better optimization the functionality of the design is partitioned into the multiple domains. To achieve the speed, area, and power requirements the design and optimization constraints need to be defined.
3. Consider the design having processors. Memory controllers, bus interface logic, DSP, and video blocks. The operating frequency of the processor is 500 MHz, DDR controller operates at 333.33 MHz, video decoder operates at 200 MHz, and the DSP processor operates at the frequency of 250 MHz. Other glue logic

and buses operate at the frequency of 150 MHz and the design is partitioned into five clock domains. In such circumstances, the design constraints to achieve the required speed, power, and area are specified in the block-level constraints.sdc file.

4. What exactly the synthesis tool does is, it uses the RTL design, constraints with the required libraries try to generate the gate-level netlist for these functional blocks. The synthesis tools are intelligent enough as they go through the various optimization phases to meet the constraints.
5. At the block level if constraints are not met, then it is essential to modify the RTL or tweak the architecture, and it can have very good impact on the later stages of the design.
6. This phase can even be helpful to understand and freeze the risks in the design (Script 9.1).

```
set active_design processor_functionality
analyze -format Verilog $active_design.v
elaborate $active_design
current_design $active_design
link
uniquify
set_wire_load_model -name SMALL
set_wire_load_mode top
set_operating_conditions WORST
create_clock -period 10 -waveform [list 0 5] master_clk
set_clock_latency 1.0 [get_clocks master_clk]
set_clock_uncertainty -setup 2.0 [get_clocks master_clk]
set_clock_transition 0.1 [get_clocks tck]
set_dont_touch_network [list tck master_reset]
set_driving_cell -cell BUFF1X -pin Z [all_inputs]
set_drive 0 [list master_clk master_reset]
set_input_delay 2.0 -clock master_clk -max [all_inputs]
set_output_delay 2.0 -clock master_clk -max [all_outputs]
set_max_area 0
set_fix_multiple_port_nets -buffer_constants -all
compile -scan
check_test
remove_unconnected_ports [find -hierarchy cell {"*"}]
change_names -h -rules BORG
set_dont_touch current_design
write_hierarchy -output $active_design.db
write_format Verilog -hierarchy -output $active_design.v
```

Script 9.1 Block-level synthesis using Synopsys DC [1]

### 9.4.1 Chip-Level Synthesis and Constraints

At the top level, the design is integration of the different functional blocks, IPs, and processors. The top-level constraints need to be derived depending on the requirement of the speed, power, and area. At the initial stages of the design cycle during the architecture level, the information about this is gathered and documented.

For example, my SOC design should work at 1 GHz operating frequency and has the multiple clock domains. So is it essential that every functional block need to be operated at the 1 GHz? Answer is no! The reason being the overall clocking rate to perform the single operation in one cycle is 1 GHz and as the design have multiple functional blocks, depending on the partition and they can be operated individually at high speed or at low speed also.

The architect and the design teams take care of the synchronization for such designs. For the functional- and timing-proven IPs, it can be possible to set the *don't touch* attributes during the iterative phase of synthesis and optimization (Script 9.2).

## 9.5 Synthesis for SOC Prototype Using FPGA

Synthesis during the prototype cycle can be performed before or after design partitioning. The synthesis is the process of converting the RTL into equivalent gate-level netlist. If we consider the ASICs, then the synthesis is the process to get the netlist using the standard cells depending on the process node. In case of FPGA, outcome of the synthesis is the netlist using the available FPGA resources. More about the FPGA architecture is discussed in Chap. 11.

Depending on the target FPGA, the synthesis tool generates the FPGA netlist and the FPGA back-end tool uses the design constraints and netlist during implementation. Most of the time during the prototype, we need to understand about the expected speed or performance for the design and that can be accomplished by few synthesis tools and it can be useful during early phase of the design cycle to tweak the architectures, constraints to achieve the required performance.

For the FPGA design, the design flow from the RTL to device programming is described in Fig. 9.4.

RTL design for the SOC prototype is one of the major milestones, and in this milestone, the RTL is tweaked to fit into the target FPGA. This may involve the changes in the IPs, clock resources, and resizing the memory blocks.

The design partitioning is required if the design specifications are complex and design does not fit in the single FPGA. If multiple FPGAs are needed to prototype and test the design, then the design can be partitioned manually or by using the partitioning tools. The design partitioning is discussed in detail in next few chapters.

For the implementation phase, we need to generate the constraints and these constraints are to define the required speed and pin placements. These can be used by the back-end tools to optimize the design. Although the vendor-dependent directives

```

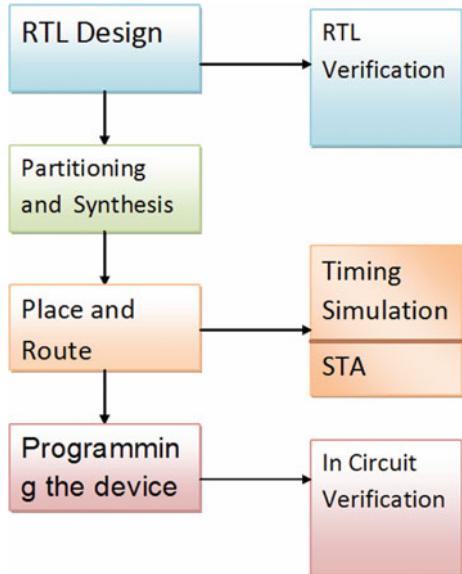
set active_design processor
set sub_modules {processor_logic decoding bus interface memory
interrupt_control}
foreach module $sub_modules {
set syn_db $module.db
read_db syn_db
}
analyze -format Verilog $active_design.v
elaborate $active_design
current_design $active_design
link
uniquify
set_wire_load_model -name LARGE
set_wire_load_mode enclosed
set_operating_conditions WORST
create_clock -period 10 -waveform [list 0 5] master_clk
set_clock_latency 1.0 [get_clocks master_clk]
set_clock_uncertainty -setup 2.0 [get_clocks master_clk]
set_clock_transition 0.1 [get_clocks master_clk]
set_dont_touch_network [list master_clk master_reset]
set_driving_cell -cell BUFX1X -pinZ [all_inputs]
set_drive 0 [list master_clk master-reset]
set_input_delay 2.0 -clock master_clk -max [all_inputs]
set_output_delay 2.0 -clock master_clk -max [all_outputs]
set_max_area 0
set_fix_multiple_port_nets -all -buffer_constants
compile -scan
remove_attribute [find -hierarchy design {"*"}] dont_touch
current_design $active_design
uniquify
check_test
create_test_patterns -sample 5
preview_scan
insert_scan
check_test
compile -only_design_rule
remove_unconnected_ports [find -hierarchy cell {"*"}]
set_dont_touch current_design
write -hierarchy -output $active_design.db
write -format Verilog -hierarchy -output $active_design.v

```

**Script 9.2** Top-level constraints script using Synopsys DC

and algorithms at the synthesis level can optimize the design to achieve the desired performance, it is essential to give the implementation constraints.

A place and route tool uses the FPGA netlist and the design constraints to place and map the design for the target FPGA. The FPGA bitstream file will be loaded into

**Fig. 9.4** FPGA design flow

the FPGA to achieve the desired functionality. Although it looks simple, this process involves multiple steps such as mapping, placement, routing, and timing analysis.

Following can be few of the back-end tools useful while prototyping the SOC

- Floor planning tool:** Consider I need to place hundreds of IOs to achieve the performance, then what I should think? I will use the floor planning tool which can achieve the IO placement, and for few critical designs, it can serve as the place the logic to meet the performance.
- Core Generators:** This kind of tool is used to infer the IP cores for the specific RTL. The objective and use of the core generator are to understand the RTL structures and replace them by the existing IP cores available for that FPGA. In layman's language, it is like replacing the RTL by its FPGA equivalent logic.
- FPGA editors:** This can be used to modify the FPGA design after the placement and routing. This can be used as debugging tool at the low level which allows the designer to modify the FPGA block placements, routing, etc.
- In circuit debugging tool:** This allows the designer to capture and view the internal design nodes. For more details refer to Chap. 16.

Now let us focus on the important practical aspect that how the logic is mapped inside the FPGA? (More details in Chap. 11)

### ***9.5.1 How Logic Is Mapped Using CLBs?***

The logic is mapped using the CLBs as each CLB consists of multiple input LUTs and registers. If we consider the Virtex-7 architecture, then using the six-input logic function, logic is mapped to have six input LUTs. For more number of inputs, the LUTs can be cascaded, and for the less number of input variables, the LUTs will be splitted to realize the function.

Consider the practical scenario where 4 or 5 inputs are used by the logic function, then it can map the logic inside one CLB.

Shift register or distributed RAM is inferred using the SLICEM.

If we use the efficient FPGA synthesis tool, then it can detect the possibility of the set/reset assertions for asynchronous and synchronous set and reset to avoid the malfunctioning inside the FPGA.

### ***9.5.2 How DSP Blocks Are Mapped?***

The Virtex-7 has the dedicated DSP block and to improve the FPGA performance, and to avoid the use of the other functional blocks, the synthesis tool can map the DSP functionality using the dedicated DSP blocks.

Consider the multiply and accumulate function (MAC) this can be mapped using the DSP block for the improved performance. If the DSP algorithm is more complex, then multiple DSP blocks can be used to implement the wider range of arithmetic algorithms. If we wish to have the pipelining in the DSP algorithm, then the pipelined logic can be mapped by using the DSP blocks.

### ***9.5.3 How Memory Blocks Are Mapped Inside FPGA?***

We need to have the internal storage in the form of the distributed RAM or block RAMs. The performance of the overall design depends upon the mapping of these blocks efficiently by synthesis tool. What we need to look into is that whether the synthesis tool infer these blocks or not?

The single- and dual-port RAMs should be inferred by the synthesis tool automatically, and for the improved design performance, the adjacent pipelined registers should be picked up automatically at the input and output. For improved speed and area, it is essential that the synthesis tool can split the large memories into the multiple BRAMs and even they should be able to have the address and data according to the functional requirements.

If we consider the synthesis tool features, then for the SOC design the synthesis tools are smart and intelligent enough due to use of the algorithms to partition and realize the design. During prototyping, the designer should take care of use of the

vendor-dependent features of such tools to achieve the desired results. Most of the time we observe that such process is automated in the industry by using the synthesis scripts.

But practically what care design team should take: Let me share my experience when I was working on one of the SOC projects during past decade!

1. The first important point I thought by visualizing the architecture that the design is too complex and needs partitioning.
2. If my SOC functionality is larger than FPGA what I should do? I need to use the multiple FPGAs.
3. Is it possible that I can achieve the desired speed of SOCs using FPGAs? Practically not because the SOC speed is faster as compared to FPGA.
4. Is it that my RTL can directly map on the FPGA? Answer is no; I need to tweak the RTL and make the changes and make it FPGA resource compliant, For example, gated clock implementation in the ASIC and FPGA differs.

## 9.6 Practical Scenarios During FPGA and ASIC Synthesis

This section describes few of the practical scenarios during the ASIC and FPGA synthesis.

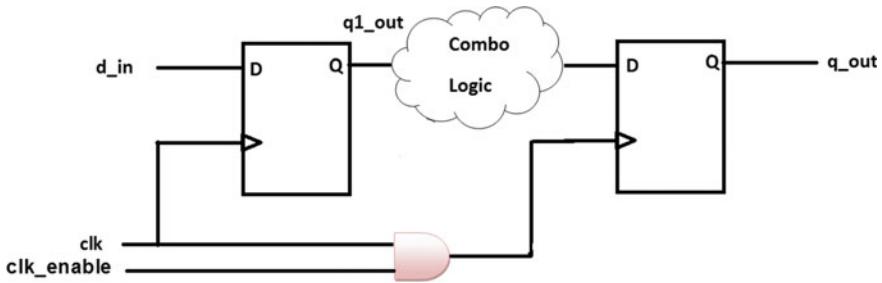
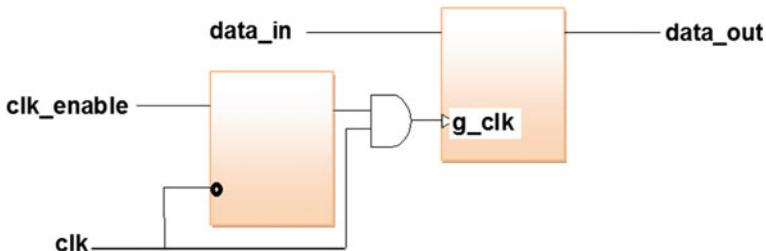
### 9.6.1 *Gated Clocks and Conversions*

The clock gating conversions can be accomplished at the RTL level as well as using the EDA tool features. Using back-end flow during the clock tree synthesis, the clock buffers can be added to balance the clock skew. The clock tree with the balanced clock skew can be routed to get the better timing and performance. But this is not possible for the FPGA design flow. This section describes the clock gating technique for the ASIC/FPGA designs.

When the design functional block needs to be inactivated then the clock can be stopped by using the clock gating mechanism. This is used to save the dynamic power. At the RTL level, this can be accomplished by using the clock and the `clock_enable` inputs and shown in Fig. 9.5.

### 9.6.2 *Gated Clock Implementation for ASIC*

For the ASIC designs, the clock gating can save the significant amount of the dynamic power. The clock gating cells are available in the library. If enable clock gating options

**Fig. 9.5** Gated clock design**Fig. 9.6** Clock gating cell

are used according to the design requirements, then these cells can be inferred by the synthesis tool.

The clock gating cell is shown in Fig. 9.6.

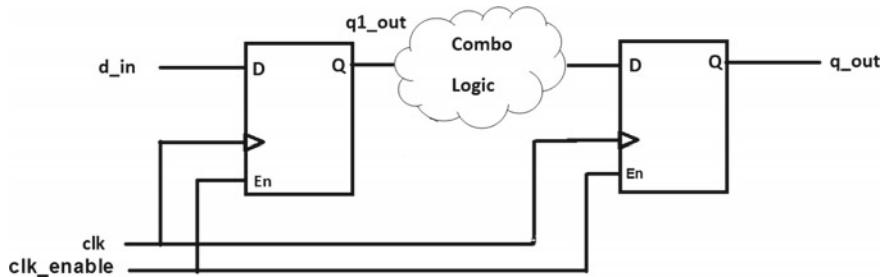
### 9.6.3 *Gated Clock Implementation for FPGA*

For the FPGA designs, the clock gating cells used in the ASIC need to be implemented at the FPGA fabric level. These cells as shown can be implemented using the LUT if and of clk and clk\_enable is used. But the issue is the glitches in the clock as AND logic switch is in the clock path. So by using vendor-specific EDA tool options, they can be implemented as shown in Fig. 9.7.

## 9.7 Important Takeaways and Further Discussions

Following are important takeaways from this chapter.

1. The ASIC synthesis infers the gate-level netlist using the ASIC cell library.
2. The FPGA synthesis infers the gate-level netlist using FPGA functional blocks such as CLBs, IOBs, DSP, clocking network, BRAMs.



**Fig. 9.7** Clock gating for FPGA

3. The synthesis tool uses the libraries, RTL design, and constraints to perform the synthesis.
4. The optimization constraints are speed, power, and area.
5. The synthesis can be performed at the block level and at the chip level.
6. The constraints can be one of the input file to synthesis tool using the (.sdc) file.
7. The design rule constraints can be max transition, max or min capacitance, and cell degradation.
8. The design partition for the larger SOC design can yield a better performance.
9. The clock gating logic for the ASIC and FPGA is different. So during the SOC prototype, it is essential to have gated clock conversion.

The next chapter will focus on the static timing analysis (STA), and how it is different for the FPGA and ASIC designs. The chapter is useful for the SOC prototyping to understand the timing and time budgeting at the different FPGA boundaries and interfaces.

## Reference

1. [www.synopsys.com](http://www.synopsys.com)

# Chapter 10

## Static Timing Analysis



*Under thermal equilibrium the product of the free electron concentration and free hole concentration is equal to a constant equal to the square of intrinsic carrier concentration.*

Mass action law for semiconductor

**Abstract** The chapter discusses the static timing analysis (STA) and the role of the STA engineer. The timing paths, maximum frequency calculations, input insertion delay, and output insertion delays are discussed in this chapter with the practical scenarios. The Synopsys PT commands are discussed in this chapter. How to achieve the timing performance to meet the timing constraints is also discussed with the practical scenarios. The chapter is useful for the ASIC and SOC designers to understand the STA concepts and techniques to overcome timing violations in the design. Even this chapter discusses the FPGA timing analysis.

**Keywords** STA · DTA · Timing paths · Reg to output · Input to output  
Reg to reg · AT · RT · Slack · Skew · Setup · Hold · Clock to q delay  
Delay derating · OCV · Dynamic simulation · Test vectors · Coverage

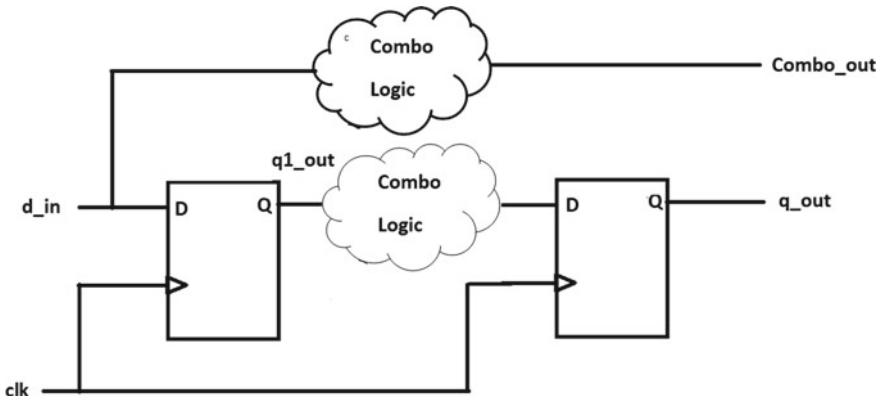
### 10.1 Synchronous Circuits and Timing

Meeting timing of the synchronous circuit is the important task, and during STA all the timing paths are analyzed by the timing analyzer. Consider the sequential synchronous circuit shown in Fig. 10.1.

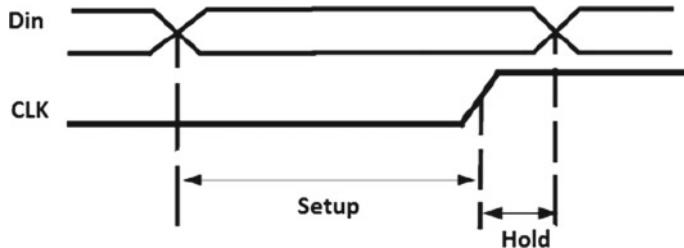
As shown in Fig. 10.1, the synchronous sequential circuit is driven by the common clock source and named as ‘clk’. The outputs are `Combo_out` and `q_out`. Input to the sequential circuit is `d_in`.

The timing parameters of the flip-flop are

- Setup time ( $t_{su}$ )
- Hold time ( $t_h$ )



**Fig. 10.1** Synchronous sequential circuit



**Fig. 10.2** Setup and hold time of flip-flop

- Propagation delay of flip-flop (Clock to q delay) ( $t_{ctoq}$  or  $t_{pdff}$ )

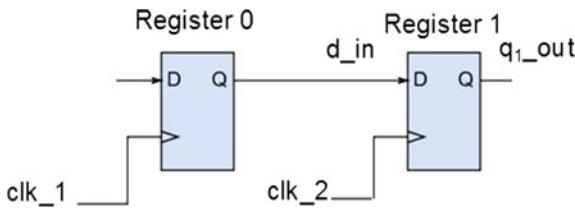
To have an understanding of these parameters, let us consider Fig. 10.2.

**Setup Time:** It is defined as the minimum amount of time for which data input (Din) of a sequential element must be stable before the arrival of the active clock edge (clock transition). In this book, the setup time is denoted by  $t_{su}$ .

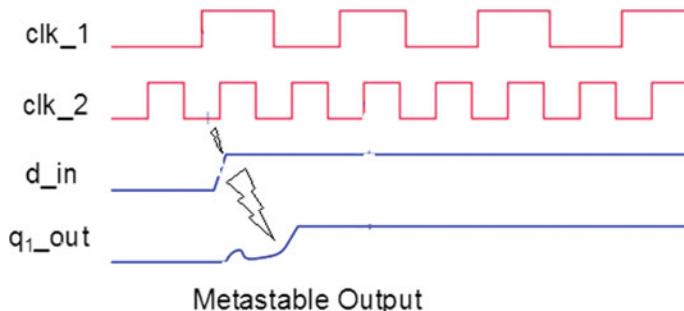
**Hold Time:** It is defined as a minimum amount of time for which the data input (Din) of the sequential device must be stable after the arrival of the active clock edge (clock transition). In this book, the hold time is denoted by  $t_h$ .

**Clock to q Delay:** It is the propagation delay of the sequential circuit element after the arrival of the active clock edge (clock transition). In this book, the flip-flop delay is denoted by  $t_{pdff}$ .

If setup or hold time is violated, then the sequential logic goes into metastable state. During timing analysis, the timing analyzer checks for all the timing paths to make sure that the timing constraints are met.



**Fig. 10.3** Design with metastable state



**Fig. 10.4** Timing sequence with metastable output

## 10.2 Metastability

As stated earlier, if any of the timing parameter is violated, then the flip-flop goes into the metastable state. Consider the scenario described in Fig. 10.3.

As shown in Fig. 10.3, the register '0' is sensitive to the rising edge of 'clk\_1' and register '1' is sensitive to the rising edge of clock source 'clk\_2'. Due to the phase difference between the 'clk\_1' and 'clk\_2', the output of register '1' goes into the metastable state. The timing sequence is shown in Fig. 10.4.

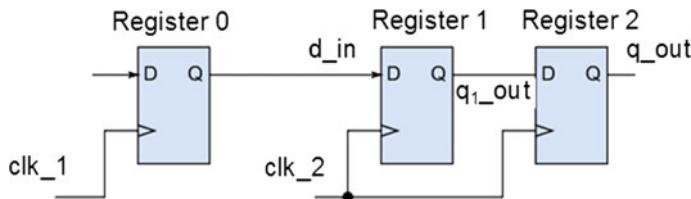
As shown in Fig. 10.4. The  $d_{in}$  input of register 1 has changed during rising edge of the  $clk_2$  and hence has the setup time violation. Under such circumstances, the register '1' goes into the metastable state.

To avoid the metastability, the multistage (multiflop) level synchronizer can be used. Figure 10.5 describes the use of the two-stage level synchronizer in the design to resolve the metastable issue.

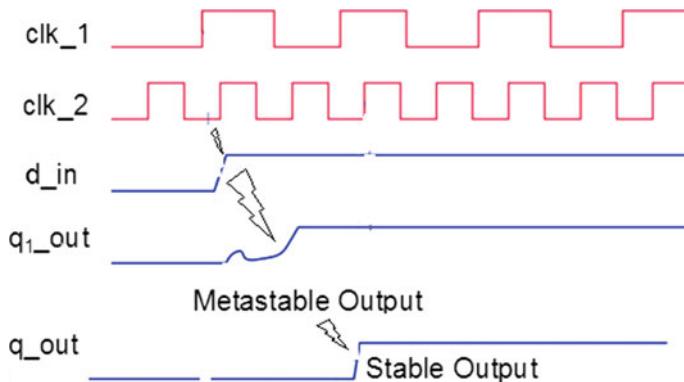
As shown in Fig. 10.5, although register '1' goes into the metastable state, on the next rising edge of the clock 'clk\_2', the output ' $q_{out}$ ' is forced into the valid state. So by adding one more registers in the output path, the metastability issue is eliminated.

Always the register '1' setup and hold parameters are violated. So during synthesis, it is essential to disable the timing from 'clk\_1' to register '1' output ' $q_{1\_out}$ '.

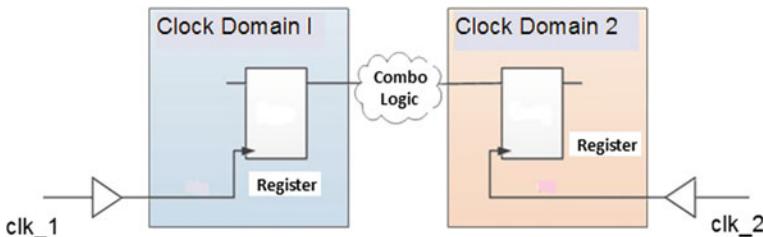
The timing sequence for sampling of the ' $d_{in}$ ' using two-stage level synchronizer is shown in the following Fig. 10.6.



**Fig. 10.5** Sampling  $d_{in}$  using two-stage level synchronizer



**Fig. 10.6** The timing sequence using two-stage level synchronizer



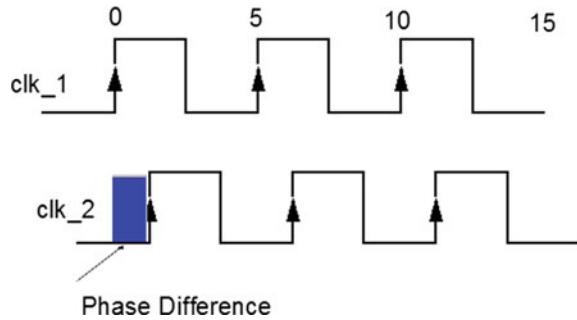
**Fig. 10.7** Multiple clock domain design

### 10.3 Metastability and Multiple Clock Domain Designs

The multiple clock domain design is shown in Fig. 10.7. As shown, two different clock domain blocks are edge sensitive to the clock sources ‘clk\_1’ and ‘clk\_2’ respectively. If clock frequency is same or different, the ‘clk\_1’ and ‘clk\_2’ may have the phase difference. Due to the phase difference between the ‘clk\_1’ and ‘clk\_2’, both clock domain logic is not triggered at the same time.

So the issue of data integrity exists while passing data between multiple clock domains. Hence, it is recommended to use the synchronizers while passing data between clock domain 1 and clock domain 2.

**Fig. 10.8** Clocks with the phase difference



The clock generation using two different clock sources with the phase difference or clock skew is shown in Fig. 10.8. As the clocks are skewed with respect to each other, the time instance at which the sequential logic in the multiple clock domain triggers is different, and hence, such type of design has issue of data integrity.

## 10.4 Timing Analysis

The era of the billion gate count design has witnessed the significant amount of changes in the process nodes. The current process node which is used is 10 nm, and even the process node will drop down to 7 and 5 nm during the next decade. But this is limited due to the physical conditions and parameters. Meeting timing for any chip is the highest priority to achieve the desired performance, and designers are spending the higher amount of efforts in addressing the design performance. Timing analysis can be of type static or dynamic.

### 10.4.1 Dynamic Timing Analysis (DTA)

As the name indicates, the dynamic simulator is used to perform the timing analysis. If the design consists of the various functional blocks, then the timing and functionality are verified by the dynamic simulator. So let us think about the requirements for the DTA!

To perform the DTA, vectors logic simulators, and the timing information is required. For the block-level timing or the chip-level timing, this methodology uses the input vectors to exercise the functional paths depending on the dynamic timing behavior. The main challenge in this type of methodology is the time required for the creation of the vectors with the high level of the test coverage. So the better and effective method is the STA as it is non-vectorized approach.

### 10.4.2 Static Timing Analysis (STA)

Traditional simulator has the limitation in the speed and capacity, and as there is limited time to market and even the complexity of the chip is high, it is better to use the STA. It is the exhaustive method of debugging, analyzing, and validating the design timing performance. In this method, firstly the design is analyzed and then all possible paths are timed.

The timed paths are checked against the timing requirements of the design. STA environment can accommodate the billion gate count design as it is non-vectorized approach. As it is not based on the functional vectors, it is very fast. Still it is very exhaustive as every path in the design is checked for the timing violations. So in the broad way we can conclude that the STA is not to verify the functionality of the design but it is for checking the timing.

As stated above the STA is used for the synchronous designs, and if the design has asynchronous blocks, then the dynamic simulation needs to be performed. Even for the mixed signal designs, the dynamic simulation can play the crucial role.

## 10.5 Timing Closure

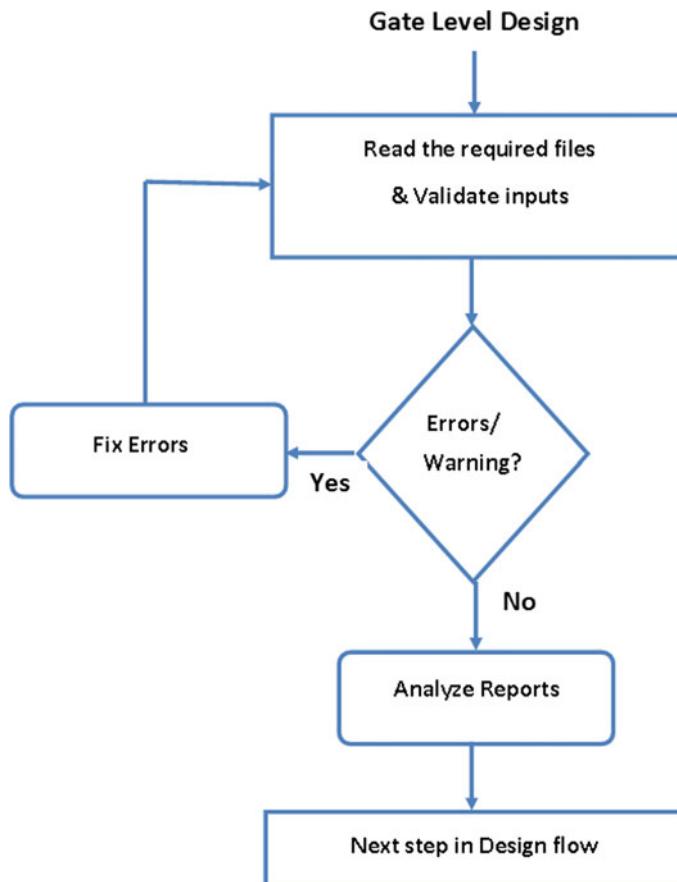
The timing closure is the ability of the timing analyzer to detect and fix the timing problems in the design as early as possible. This can be accomplished by using the STA and also by dynamic simulation with the SDF back annotation.

If the timing is failed means the timing goals are not achieved, then the resynthesis, performance improvements, micro-architecture tweaking, timing constraint modifications, and in worst case, the redesign need to be done, and it is iterative process.

Figure 10.9 gives information about the flow used by the timing analyzer.

### 10.5.1 STA Important Steps

1. Break down the design into different timing paths
  - a. Input-to-register timing path
  - b. Register-to-register timing path
  - c. Register-to-output timing path
  - d. Input-to-output timing path.
2. Calculate the delay of each timing path
3. Check the delay of each timing path against the timing constraints. If they are met, then no timing violation.

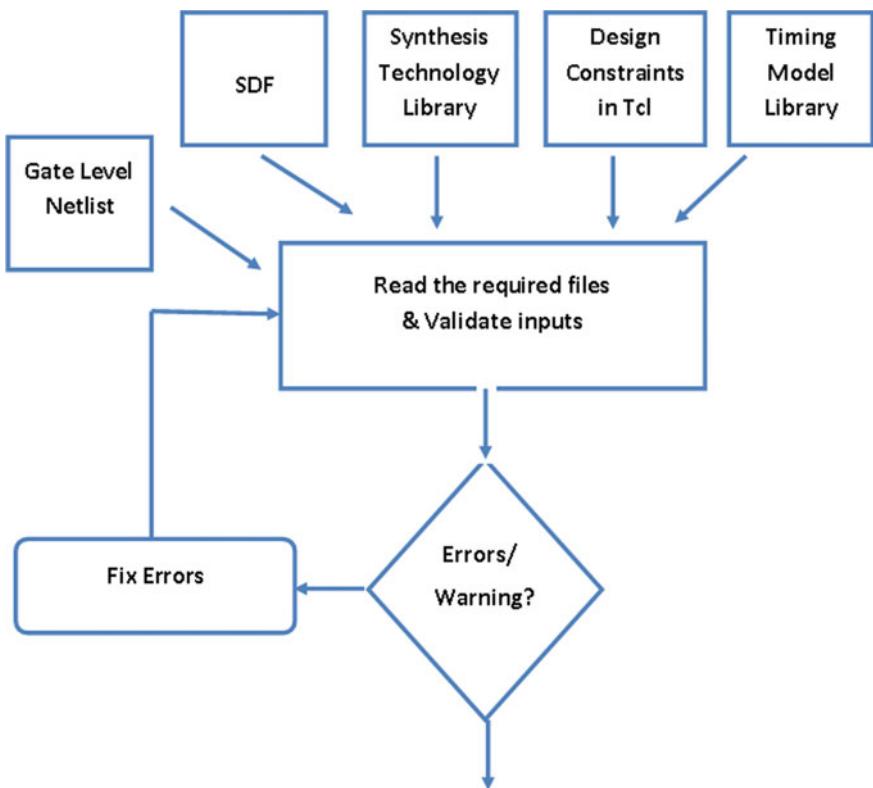


**Fig. 10.9** Timing analyzer flow

As discussed earlier, STA is popular and efficient methodology of the timing analysis as it is faster as compared to dynamic simulation. Block-level as well as full chip timing can be checked using the STA as it has exhaustive timing coverage. And another important point is, there is no any need of the vectors while performing timing analysis. Still this methodology has disadvantage as it reports false paths.

Figure 10.10 gives information about the various inputs used by the timing analyzer. At higher level, we can say that to perform STA following is required

- Gate-level netlist
- Constraints (.SDC)
- Extracted nets (SPEF)
- Libraries (.lib)



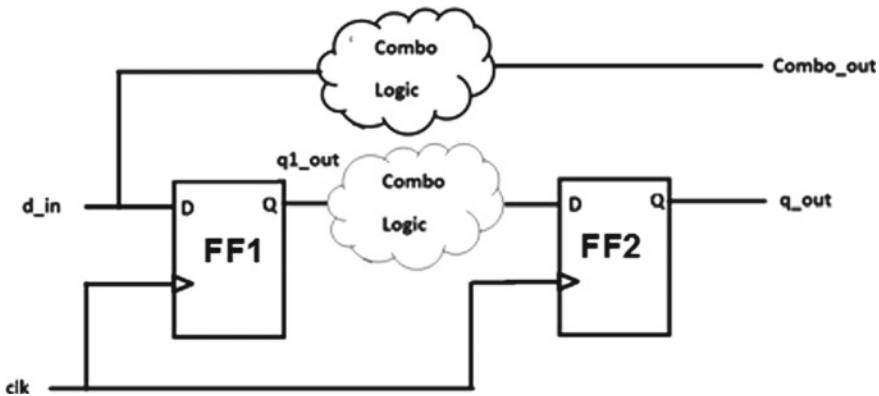
**Fig. 10.10** Inputs required for STA

## 10.6 Timing Paths in the Synchronous Design

As discussed in the above section, the synchronous circuit has four timing paths and with reference to the Figure 10.11 they are explained in this section.

To identify the timing paths, consider the start point and endpoint. Start points are input port or clock port, and endpoints are data input of sequential element or output port.

- **Input-to-Register Path:** The path from the primary input to data input of the flip-flop 1 (FF1) is input-to-register path.  
Consider start point  $d_{in}$  and endpoint  $D$  of FF1. So  $d_{in}$  to FF1/D is input-to-register path also called as input-to-reg path.
- **Register-to-Register Path:** The path from clock port of FF1 to data input of flip-flop 2(FF2) is called as register-to-register path. FF1/clk to FF2/D. This path is decisive factor to find the maximum operating frequency for the design.
- **Register-to-Output path:** The path from clock port of FF2 to the output port of synchronous sequential circuit is called as register-to-output path.



**Fig. 10.11** Synchronous circuits and timing paths

**Table 10.1** Synopsys PT commands to define clock and delay

Timing goal	Command
Define clock period	<code>create_clock</code>
Define input delay	<code>set_input_delay</code>
Define output delay	<code>set_output_delay</code>
Define clock skew	<code>set_clock_skew</code>

As shown, consider the start point as clock port of the **FF2** and endpoint as output port of `q_out`. So register-to-output timing path is **FF2/clk** to **FF2/q\_out**.

- **Input-to-Output path:** It is also called as combinational path, and it is path from the input port to output port.

Consider the start point as input port `d_in` and endpoint as output port `Combo_out`, then the input-to-output path is `d_in` to `Combo_out`

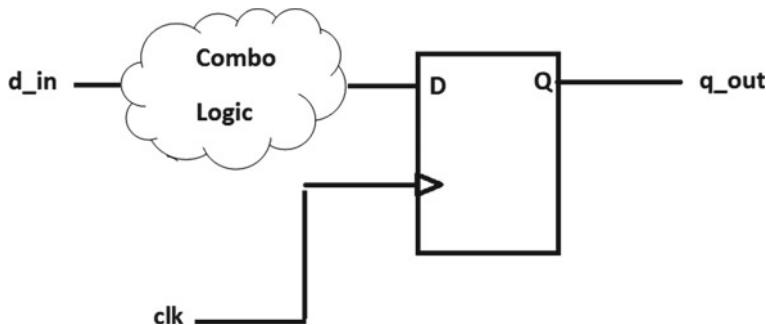
Table 10.1 gives information to specify the timing goals for the synchronous sequential circuit.

The timing paths are described in Sect. 10.6.1 with more details.

### 10.6.1 Input-to-Register Path

As discussed earlier, the input-to-register path is from `d_in` to `D` of the sequential element. So the timing analyzer checks that the timing in this path is met or not? For the desired operation; the output of the **Combo Logic** must be stable and should not violate the setup time. That is, `D` input should be stable at or before  $t_{clk} - t_{su}$ . It is shown in Fig. 10.12.

To define the input delay, use the following command



**Fig. 10.12** Input-to-register path

Command	Description
<code>set_input_delay -clock &lt;clock_name&gt; &lt;input_delay&gt; &lt;input_port&gt;</code>	Used to define the input port delay with reference to the clock. To define 1 ns delay, use the command <code>set_input_delay -clock clk 1 d_in</code>

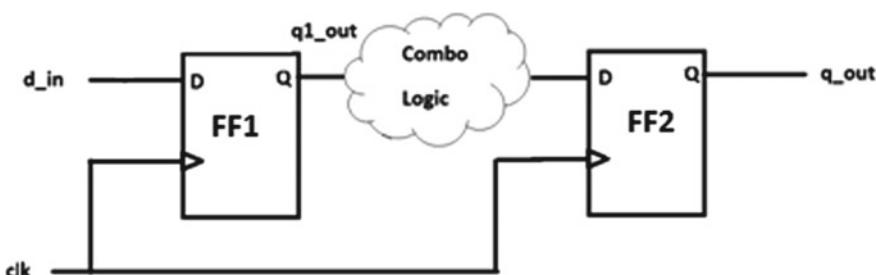
### 10.6.2 Register-to-Register Path

As discussed earlier, the timing path from start point FF1/clk to the FF2/D is called as register-to-register timing path. Consider Fig. 10.13.

Let us think what is requirement in this path so that timing should be met? The slack which is the difference between the data required time and data arrival time should be positive. This indicates that the input `D` of FF2 should be stable before arrival of the active edge of clock.

$$\text{Setup Slack} = \text{Data Required Time (RT)} - \text{Data Arrival time (AT)}$$

$$RT = t_{clk} - t_{su}$$



**Fig. 10.13** Register-to-register timing path

$$AT = t_{pdff1} + t_{combo}$$

So RT should be greater than or equal to AT; then only setup slack is positive. The time period of clock can be found by using

$$\begin{aligned}t_{clk} - t_{su} &= t_{pdff1} + t_{combo} \\t_{clk} &= t_{pdff1} + t_{combo} + t_{su}\end{aligned}$$

Thus, the maximum operating frequency should be  $f_{max} = (1/(t_{pdff1} + t_{combo} + t_{su}))$ . If we consider the  $t_{pdff1} = 2$  ns,  $t_{combo} = 2$  ns, and  $t_{su} = 1$  ns, then the maximum operating frequency should be  $(1/(2+2+1))\text{ns} = 200$  MHz.

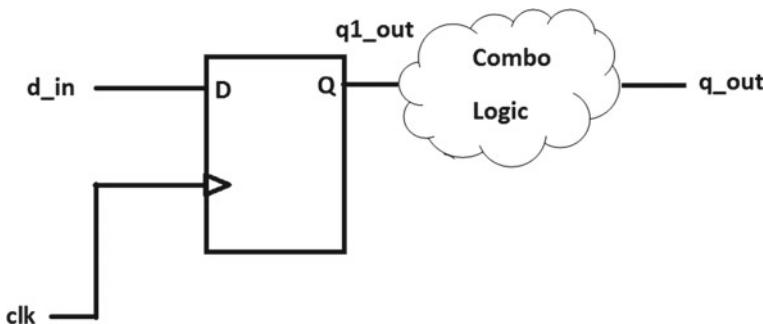
For the hold check, the slack is defined as difference between the data arrival time and data required time, and it should be positive. That is, the data should not arrive very fast and should not violate the hold time.

Setup and hold slack calculations and the timing reports are discussed in the next few sections.

### 10.6.3 Register-to-Output Path

As discussed earlier, the register-to-output path is from clk port to q\_out of the sequential element. So the timing analyzer makes sure that the timing in this path is met or not. For the desired timing, the output of the Combo Logic should be stable and should not violate the setup time. That is D input should be stable at or before  $t_{clk}-t_{su}$  (Fig. 10.14)

To define the output delay, use the following command



**Fig. 10.14** Register-to-output path

Command	Description
<code>set_output_delay -clock &lt;clock_name&gt; &lt;input_delay&gt; &lt;input_port&gt;</code>	Used to define the output port delay with reference to the clock. To define 1 ns delay, use the command <code>set_output_delay -clock clk 1 d_in</code>



**Fig. 10.15** Input-to-output path

#### 10.6.4 Input-to-Output Path

As discussed earlier, the input-to-output path is from input port to output port of the combinational element. The path is unconstrained path as the output is function of the present input only. This is combinational path and shown in Fig. 10.15.

### 10.7 What Timing Analyzer Should Perform?

Timing analyzer should perform following main tasks for each corner and mode while performing the timing checks

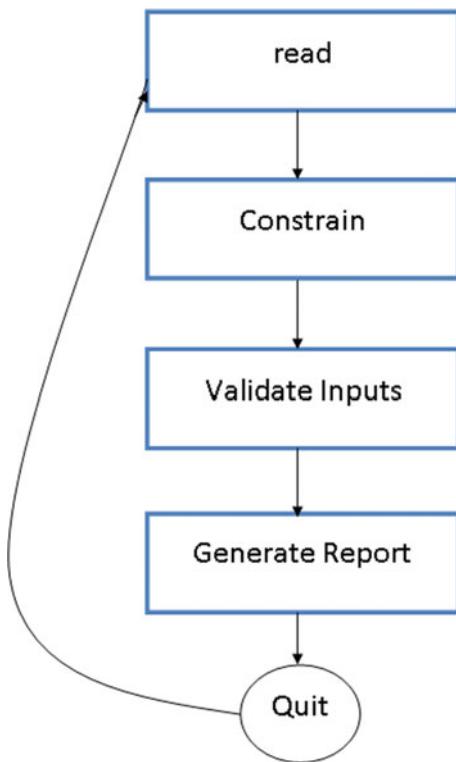
1. It should read all the required input files.
2. It should use the constraint file to check for the timing information.
3. It should validate the necessary inputs.
4. It should generate the necessary timing reports to indicate the violations in the design (Fig. 10.16).

### 10.8 Setup Time Analysis

The section discusses the setup analysis. What the timing analyzer does while performing the setup time analysis? Consider Fig. 10.17.

During the setup time analysis, depending on the operating conditions the timing analyzer performs the analysis of the timing path. The slack which is the difference between the RT and AT is computed. The positive slack indicates no setup violation in this path (Fig. 10.18).

**Fig. 10.16** Tasks performed by timing analyzer

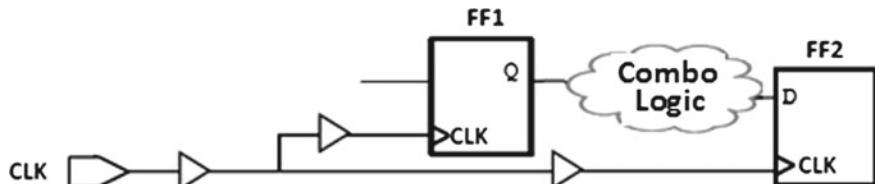


The timing analysis tool uses different modes such as

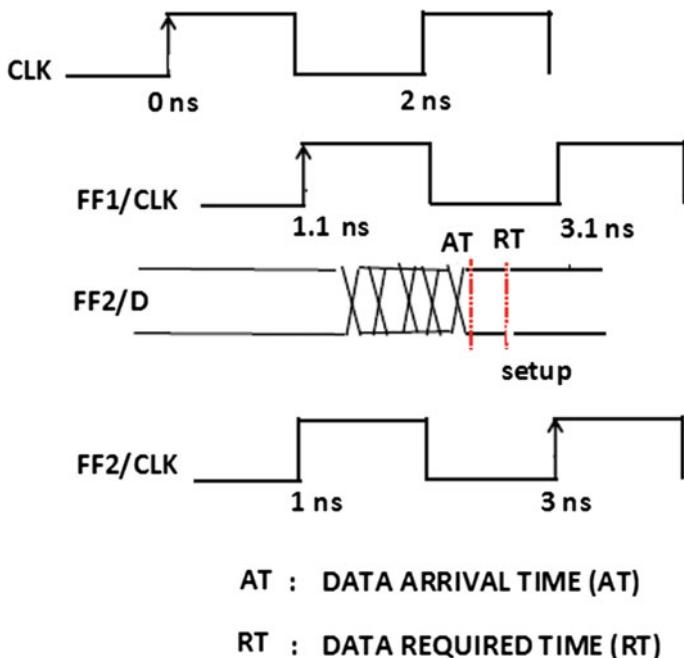
1. Single operating conditions
2. Best-case and worst-case modes
3. On-chip variation mode

Refer Table 10.2 for the different analysis modes during the setup time check.

The script using the Synopsys PT commands for setup time check is shown in Script 10.1.



**Fig. 10.17** Synchronous design



**Fig. 10.18** Timing sequence to indicate AT and RT (slack positive)

**Table 10.2** Setup time analysis [1]

Analysis mode	Data path	Launch path	Capture path
Single operating condition	Maximum delay without derating	It should be late clock, and without derating the delay should be maximum in the clock path	It should be early clock, and without derating the delay should be minimum in the clock path
bc_wc mode	Consider maximum delay, worst-case operating conditions, and late derating	Late clock and late derating the delay should be maximum in the clock path and worst-case operating conditions	Early clock and early derating the delay should be minimum in the clock path and worst-case operating conditions
On-chip Variation Mode	Consider maximum delay, worst-case operating conditions, and late derating	Late clock and late derating the delay should be maximum in the clock path and worst-case operating conditions	Late clock and late derating the delay should be maximum in the clock path and best-case operating conditions

**Table 10.3** Hold time analysis [1]

Analysis mode	Data path	Launch path	Capture path
Single operating condition	Minimum delay without derating	It should be early clock, and without derating the delay should be minimum in the clock path	It should be late clock, and without derating the delay should be maximum in the clock path
bc_wc mode	Consider minimum delay, best-case operating conditions, and late early derating	Early clock and early derating the delay should be minimum in the clock path and best-case operating conditions	Late clock and late derating the delay should be maximum in the clock path and best-case operating conditions
On-chip Variation Mode	Consider minimum delay, best-case operating conditions, and early derating	Early clock and early derating the delay should be minimum in the clock path and best-case operating conditions	Late clock and late derating the delay should be maximum in the clock path and worst-case operating conditions

```

set active_design processor
read_db -netlist_only $active_design.db
current_design $active_design
set_wire_load_model -name large
set_wire_load_mode top
set_operating_conditions WORST
set_load 40.0 [all_outputs]
set_driving_cell -cell BUFLX-pinZ [all_inputs]
create_clock -period 10 -waveform [0 5] master_clk
set_clock_latency 1.0 [get_clocks master_clk]
set_clock_transition 0.1 [get_clocks master_clk]
set_clock_uncertainty 2.0 -setup [get_clocks master_clk]
set_input_delay 2.0 -clock master_clk [all_inputs]
set_output_delay 2.0 -clock master_clk [all_outputs]
report_constraint -all_violators
report_timing -to [all_registers -data_pins]
report_timing -to [all_outputs]
write_sdf -contextverilog -output $active_design.sdf

```

**Script 10.1** Setup check [1]

The timing report is shown in Fig. 10.19. As shown, the difference between the data required time and data arrival time is positive. This indicates no timing violation due to positive slack.

If slack is not met then what need to do

1. Consider the late arrival signals in the data path.
2. Tweak the RTL
3. Tweak the architecture micro-architecture

Point	Incr	Path
clock Clk (rise edge)	0.00	0.00
clock network delay (propagated)	1.10 *	1.10
FF1/CLK (fdef1a15)	0.00	1.10 r
FF1/Q (fdef1a15)	0.50 *	1.60 r
U2/Y (buf1a27)	0.11 *	1.71 r
U3/Y (buf1a27)	0.11 *	1.82 r
FF2/D (fdef1a15)	0.05 *	1.87 r
data arrival time		1.87
clock Clk (rise edge)	4.00	4.00
clock network delay (propagated)	1.00 *	5.00
FF2/CLK (fdef1a15)		5.00 r
library setup time	-0.21 *	4.79
data required time		4.79
data required time		4.79
data arrival time		-1.87
slack (MET)		2.92

Fig. 10.19 Timing report for the setup analysis [1]

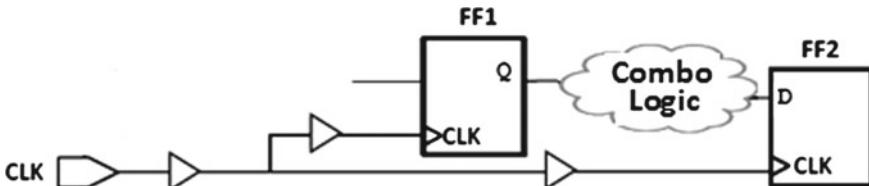


Fig. 10.20 Synchronous design

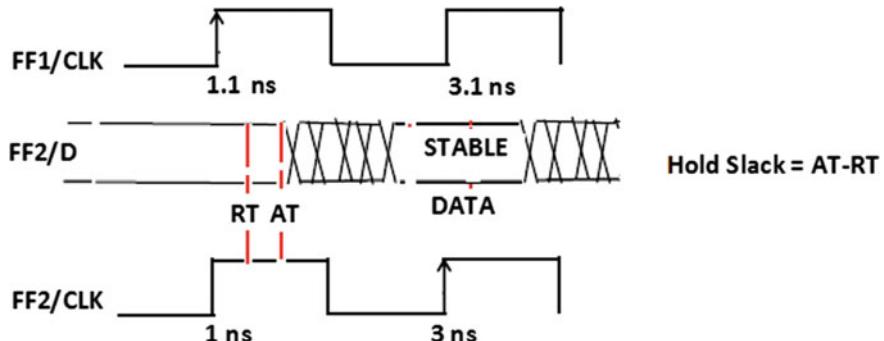
## 10.9 Hold Time Analysis

The section discusses the hold analysis. What the timing analyzer does while performing the hold time analysis? Consider Fig. 10.20.

During the hold time analysis, depending on the operating conditions the timing analyzer performs the analysis of the timing path. The slack which is difference between the AT and RT is computed. The positive slack indicates no hold time violation in this path (Fig. 10.21).

Refer Table 10.3 for the different analysis modes during the hold time check.

The script using the Synopsys PT commands for hold time check is shown in Script 10.2.



**Fig. 10.21** Timing sequence to indicate AT and RT (slack positive)

```

set active_design processor
read_db -netlist_only $active_design.db
current_design $active_design
set_wire_load large
set_wire_load_mode top
set_operating_conditions BEST
set_load 50.0 [all_outputs]
set_driving_cell -cell BUFLX -pin Z [all_inputs]
create_clock -period 10 -waveform [0 5] master_clk
set_clock_latency 1.0 [get_clocks master_clk]
set_clock_transition 0.1 [get_clocks master_clk]
set_clock_uncertainty 0.5 -hold [get_clocks master_clk]
set_input_delay 0.0 -clock master_clk [all_inputs]
set_output_delay 0.0 -clock master_clk [all_outputs]
report_constraint -all_violators
report_timing -to [all_registers -data_pins] -delay_type min
report_timing -to [all_outputs] -delay_type min
write_sdf -context verilog -output $active_design.sdf

```

**Script 10.2** Hold time check [1]

The timing report is shown in Fig. 10.22. As shown the difference between the data arrival time and data required time is positive. This indicates no timing violation due to positive slack (Fig. 10.22).

If slack is not met then what need to do

1. Consider the early arrival signals in the data path.
2. Tweak the RTL
3. Tweak the architecture and micro-architecture

Startpoint: FF1 (rising edge-triggered flip-flop clocked by Clk)		
Endpoint: FF2 (rising edge-triggered flip-flop clocked by Clk)		
Path Group: Clk		
Path Type: min		
Point	Incr	Path
clock Clk (rise edge)	0.00	0.00
clock network delay (propagated)	1.10 *	1.10
FF1/CLK (fdef1a15)	0.00	1.10 r
FF1/Q (fdef1a15)	0.40 *	1.50 f
U2/Y (buf1a27)	0.05 *	1.55 f
U3/Y (buf1a27)	0.05 *	1.60 f
FF2/D (fdef1a15)	0.01 *	1.61 f
data arrival time		1.61
clock Clk (rise edge)	0.00	0.00
clock network delay (propagated)	1.00 *	1.00
FF2/CLK (fdef1a15)		1.00 r
library hold time	0.10 *	1.10
data required time		1.10
data required time		1.10
data arrival time		-1.61
slack (MET)		0.51

Fig. 10.22 Timing report for the hold analysis [1]

## 10.10 Clock Network Latency

If we consider any SOC chip, then the clock network latency and clock distribution decide the performance of any synchronous design. The PLL is used as clock source, and during STA it is essential to define the clock source and clock network latency. The figure shows both the latencies. (Fig. 10.23).

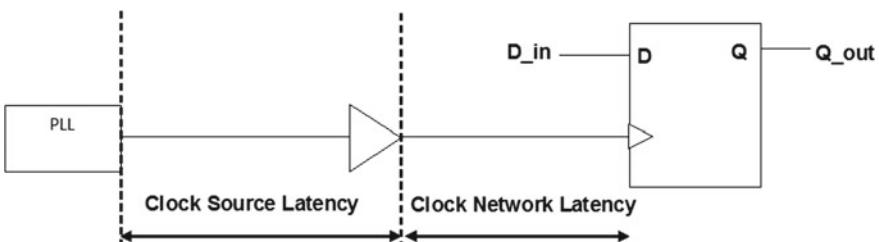


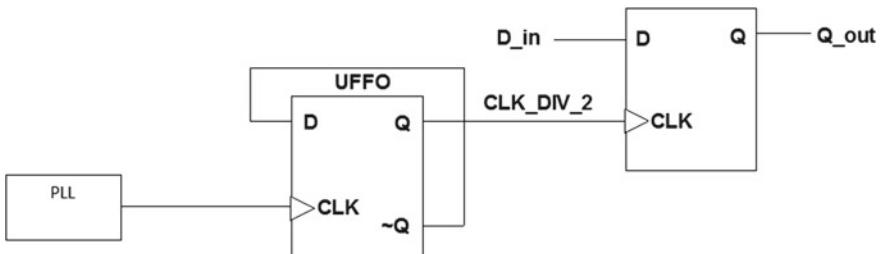
Fig. 10.23 Latencies in design

## 10.11 Generated Clock

The generated clocks in the SOC can be used as clocking source to the sequential blocks. The clocks are generated by using the clock divider networks. Figure 10.24 shows the generated clock using the clock divider. The useful Synopsys PT commands are described in Table 10.4.

## 10.12 Clock Muxing and False Paths

Most of the times we need to have clock multiplexing. The minimum and maximum frequency clocks can be used in the design depending on the design requirements. During the ASIC testing, the minimum frequency clock can be used. The false paths between these clocks need to be reported to the timing analyzer. To set the false path, use the command shown in Table 10.5 and Fig. 10.25.



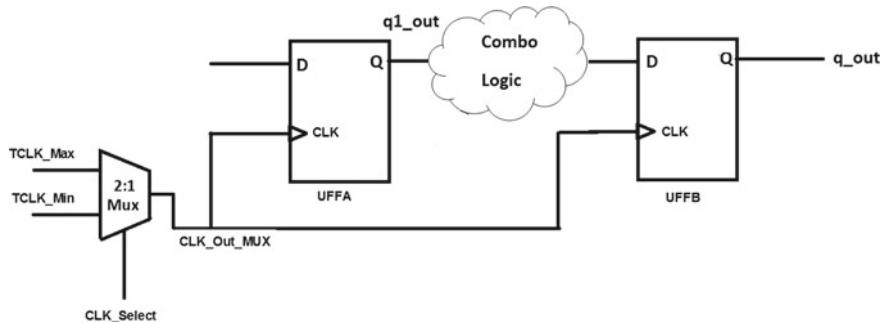
**Fig. 10.24** Generated clock

**Table 10.4** Clock and generated clock commands

Command	Description
<code>create_clock -period 10 waveform {0 5} [get_ports clk_PLL]</code>	Used to define clock having period 10 ns. The rising edge at 0 ns and falling edge at 5 ns
<code>Create_generated_clock -name CLK_DIV_2 -source UPLL0/clkout -divide_by 2 [get_pins UFF0/Q]</code>	Generated clock CLK_DIV_2 at Q

**Table 10.5** Commands to set the false paths

Command	Description
<code>set_false_path -from [get_clock Tclk_max] -to [get_clocks Tclk_min]</code>	Used to set the false path between the Tclk_max and Tclk_min
<code>set_false_path -through [get_pins UMUX/clk_select]</code>	To set the false path with respect to clk_select



**Fig. 10.25** Clock muxing and false path

## 10.13 Clock Gating

The clock gating checks need to be performed by the timing analyzer and the command described in Table 10.6 and Fig. 10.26.

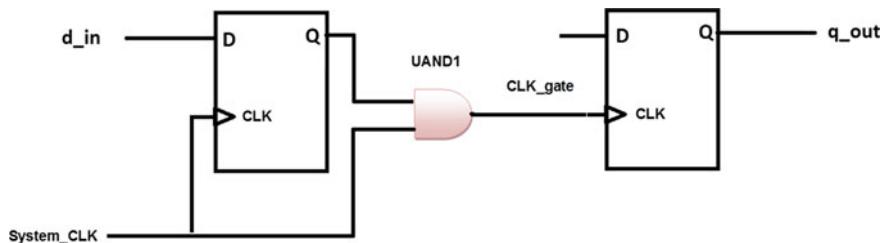
## 10.14 Multicycle Paths

The multicycle paths in the design need to be reported. The paths can be set so that the timing analyzer can perform the setup and hold check (Fig. 10.27).

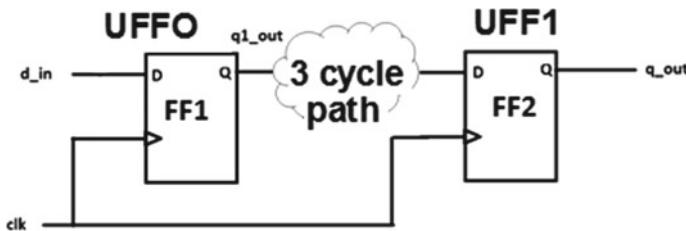
The commands used to set the multicycle path are listed in Table 10.7.

**Table 10.6** Clock gating checks commands

Command	Description
create_clock -period 10 [get_ports System_CLK]	To create the system clock of period 10 ns
create_generated_clock -name -divide_by 1 System_CLK [get pins UAND1/Z]	To get the same clock CLK_gate



**Fig. 10.26** Clock gating

**Fig. 10.27** Multicycle path**Table 10.7** Multicycle path commands

Command	Description
create_clock -name clk_master -period 10 [get_ports clk_master]	To create the master clock of period 5 ns
set_multicycle_path 3 -setup -from [get_pins UFFO/Q] -to [get_pins UFF1/D]	This set the multicycle path of 3 cycles.
set_multicycle_path 2 -hold -from [get_pins UFFO/Q] -to [get_pins UFF1/D]	This is used to move the hold check to 2 clock cycles as setup is check at 3 clock cycle.

## 10.15 Timing for FPGA Designs

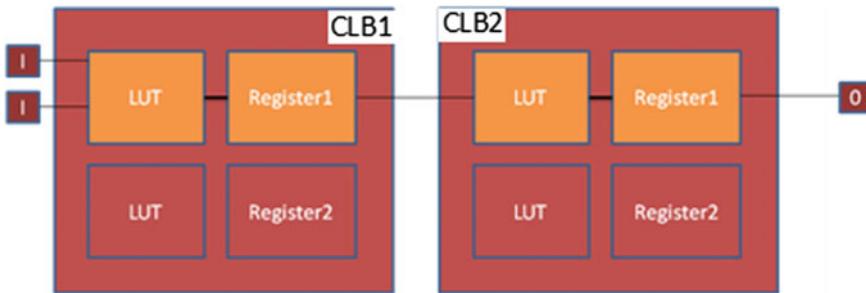
For any FPGA design, the timing analysis can be performed by using the timing analyzer, and the timing reports are useful to find out whether the setup and hold slack is met or not?

As a prototype engineer, what we need to think about is the partitioning of the functional blocks across the FPGA fabric. The better partitioned design can yield into the better timing and performance. Let us consider the SOC design which has multiple clock domains and multiple power domains. For such kind of the design, the time budgeting is important, and it can be performed at the block-level designs and at the chip level.

Most of the complex designs do not map into the single FPGA design; under such circumstances, we need to have the partitioning across multiple FPGA and then the timing analysis can be performed.

*What we need to consider?*

1. The combinational delay (LUT Delay), the setup, hold, and the flip-flop propagation delay.
2. The setup slack is defined as slack = RT - AT where RT > AT.
  - a.  $AT = t_{pdff} + t_{combo}$
  - b.  $RT = t_{clk} - t_{su}$
3. The hold slack is defined as slack = AT - RT where RT < AT.



**Fig. 10.28** Inside FPGA reg to reg path

- a.  $AT = t_{pdff} + t_{combo}$
  - b.  $RT = t_{clk} - t_h$
4. For the multiple FPGA design, we need to consider the pad delay and onboard delay.

## 10.16 Timing Analysis for the FPGA Designs

The design using single FPGA in which the logic is mapped using the CLBs and other FPGA functional blocks. The operating frequency of such designs is based on the register-to-register path delay, and it can be limited due to the IO delay while transferring the data from FPGA to other associated devices.

Consider the following figure in which the logic is implemented using the multiple CLBs, and the delay between the register is LUT delay. The CLB1 and CLB2 connections are shown in the (Fig. 10.28).

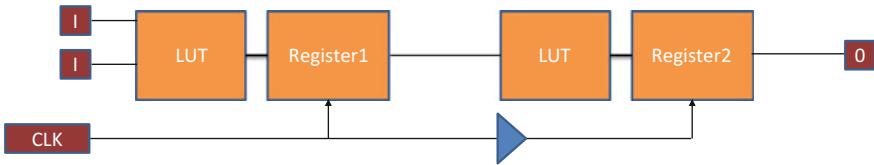
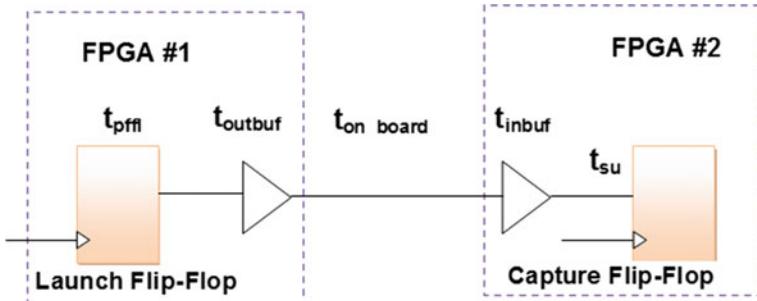
So the data arrival time from CLB1 to CLB2 is register1 delay + LUT delay in the II CLB. And data required time is  $T_{clk} - t_{su}$

$$F_{max} = (1/(t_{pdff1} + t_{lut} + t_{setup}))$$

The timing analyzer finds the register-to-register path timing, and depending on the timing constraints the slack is computed.

## 10.17 How This Discussion Is Useful During Prototyping?

The above discussion can give two important aspects and can be useful in the SOC prototyping. For more details, refer Chaps. 11–15

**Fig. 10.29** Clock buffer delay**Fig. 10.30** Direct connection between two FPGA

- For the single FPGA design, the logic is mapped on the FPGA fabric, and the operating frequency of the design is based on the critical path in the design. For the single clock domain design, if we visualize Fig. 10.29, then we can find operating frequency for the design using

$$f_{\max} = (1/(t_{pdff} + t_{lut} + t_{setup} - t_{buf}))$$

- For the design using multiple FPGA as shown, it is essential to consider the logic delays inside FPGA (on-chip delay) and onboard delay. So the maximum operating frequency of such design is less as compared to the single FPGA design (Fig. 10.30).

$$\begin{aligned} t_{clk} &= t_{pdff} + t_{outbuf} + t_{on\_board} + t_{inbuf} - t_{su} \\ f_{\max} &= 1/(t_{pdff} + t_{outbuf} + t_{on\_board} + t_{inbuf} - t_{su}) \end{aligned}$$

## 10.18 Important Takeaways and Further Discussions

As discussed in this chapter, the STA is non-vectored approach to find whether all the timing paths are met or not? Following are key important takeaway to conclude this chapter

1. STA does not need the input vector.
2. STA tool uses the netlist, constraints, libraries, and SPEF.
3. The timing paths are input to reg, reg to reg, output to reg, and input to output.
4. Input-to-output path is also called as combinational path.
5. The positive setup and hold slack indicate the timing is met.
6. The maximum operating frequency for the design is based on the register-to-register path timing.
7. For the single FPGA design, the speed of the design is based on the LUT delay, flip-flop propagation delay, and the setup time of flip-flop.
8. The clocking skew need to be considered to find out the speed of the design.
9. For the multiple FPGA design, the speed of the design is dependent on the interconnect delay.

The next chapter discusses the SOC prototyping using FPGAs and useful to understand the role of the FPGA in the SOC prototyping.

## Reference

1. [www.synopsys.com](http://www.synopsys.com)

# Chapter 11

## SOC Prototyping



*SOC prototyping using high-density FPGA is useful to detect the early bugs and to reduce risks in the SOC design*

**Abstract** The chapter discusses the SOC prototyping using FPGAs. The FPGA functional blocks are discussed in this chapter with their use. The logic inference using FPGA is discussed with the real-life scenarios. The chapter discusses the prototyping challenges and how to overcome them. The chapter is useful to understand the SOC prototyping basics and logic inference using FPGAs.

**Keywords** SOC · FPGA · CLB · LUT · BRAM · Routing · Register · PCI Clocking · Interconnect · DSP · In-built monitors

### 11.1 SOC Prototyping Using FPGA

If we observe the evolution of the FPGA architectures during this decade, then we can conclude the following points:

1. The FPGA architecture is complex and can be used for the SOC prototyping.
2. High-density FPGAs have the hard processor cores and other high-speed interfaces.
3. The million gate SOCs can be validated using the multiple FPGAs in the system.
4. The high clocking rate allows the higher speed for the prototype.
5. The FPGA IP equivalent can be integrated with the other components, and the proof of concept can be validated.
6. The turnaround time reduces, and hence, the time to market the product drops down significantly.
7. The early bugs at the implementation levels can be detected and fixed to avoid the respin of ASIC/SOC.

As the process node has shrunk to 10 nm, the complexity of design, the design risk, and the development time has grown significantly. The main challenge for every organization is to develop the low-cost products having complex functionality in small silicon area. In such circumstances, the designers are facing several development and verification challenges. To cope up with these challenges, the high density FPGAs can be used to prototype the ASIC/SOC as it reduces the overall risk.

The verified and implemented design using high density FPGAs can be resynthesized using standard cell ASIC using the same RTL (with the FPGA compatible tweaks), constraints, and scripts. There are EDA tools available to port an FPGA prototype on structured ASICs. This really reduces the overall risk in ASIC design and saves money and even reduces time to market for the product.

The following are key advantages of ASIC/SOC prototyping using FPGAs

1. **Low investment:** The shrinking process node and chip geometries involve the investment in millions of dollars in the early stage of design. Using FPGAs, the investment risk reduces.
2. **Accommodate the changes in the design:** Due to uncontrolled market conditions, there is risk involved in the design and development of products. FPGA prototype reduces such risk as the product specifications and design can be validated depending on the functional requirements or changes.
3. **System-level bug detection:** FPGA prototyping is efficient as the bugs those were not detected in simulation can be addressed and covered during prototyping.
4. **Functional bugs and fixes at early stage:** Full system verification using FPGA prototype can detect the functional bugs in the early stage of the design cycle.
5. **EDA tools and costing:** FPGA prototyping saves millions of dollar of EDA tool cost and even it saves the millions of dollar engineering efforts before ASIC tape-out.
6. **Reduced time to market:** As design using FPGA can be migrated using the EDA tool chains onto the ASICs, it saves the time to market the product with intended functionality.
7. **Shorter design cycle:** Multiple IPs can be integrated, and design functionality can be verified and tested and that can speed up the overall delivery of product to end clients or to customers.
8. **Design partitioning and validation:** Most of the cases the hardware-software portioning is visualized at higher abstraction level. The hardware-software code-sign can be evaluated at the hardware level, and it is a important milestone in the overall design cycle. So the ASIC prototyping can be useful in the tweaking of the architecture to improve the design performance. For example, if there is additional design overhead in the hardware then the design architecture can be changed by pushing few blocks in software and vice versa. This will give the more efficient architecture and design.

Table 11.1 gives information about the pros and cons of FPGA and ASIC.

The ASIC prototyping is basically the design validation of idea to check for the early functional and feasibility of the designs. The design migration from ASIC to FPGA involves the flow from RTL design to implementation and may be useful in the upgradation of design with additional features.

**Table 11.1** Comparison of FPGA versus ASIC implementation [3]

	FPGA	Hard copy	Structured ASIC	Standard cell ASIC
NRE, mask, and EDA tools	Up to a few thousand US\$, so the overall cost is low	Couple of hundred thousand US\$ for FPGA conversion and masks. So the overall cost is moderate	A couple of hundred thousand US\$ for interconnect/meta one mask so the overall cost is moderate	A million US\$ depending on the design functionality. So the cost is high
Unit price	High	Medium-low	Medium-low	Low
Time to volume	Immediate	Almost around 8–10 weeks. The additional conversion time may require for other structured ASIC products	Almost around 8–10 weeks. The additional conversion time may require for other structured ASIC products	Almost around 18 weeks + conversion time of another 18 weeks
Engineering resources and cost	Minimum	Minimal from developers but other structured products may require the additional engineering resources	Nominal but for the other structured ASIC products may require the additional engagement of the resources	High as most of the work requires the development from scratch and requires good support from back-end team
FPGA prototype correlation	Same device	For hard copy structured ASIC: Nearly identical—same logic elements, process, analog components, and packages	It depends upon the type of IP used and the functionality. Same RTL but potentially different libraries, process, analog, and packages	Same RTL but potentially different libraries, process, analog, and packages

The following are the key points need to be considered during ASIC prototyping and design migration using high-end FPGA.

- Select the suitable board:** Use the universal prototype board as it saves the time of almost 4 months to 12 months for the high-speed prototyping development.
- Partition the design for the better prototype:** Choose the FPGA device depending on the design functionality and gate count. It may not be possible to fit whole ASIC into single FPGA even if we use the high-end families of Intel FPGA or XILINX FPGAs. So the practical solution is use of multiple FPGAs. But the real issue is the design partitioning and the intercommunication between multiple FPGAs.

If the design is well defined and partitioned properly, then the manual partitioning into multiple FPGA can give the efficient results. If the design has high density and has complex functionality, then the use of automatic partitioning can play an efficient role and can result into the efficient prototype.

3. **Get the FPGA equivalent of the ASIC design:** As the design library for ASIC and FPGA is totally different, the key challenge is to map the primitives. So it is essential to map the directly instantiated primitives during synthesis, and during the implementation level that is post-synthesis, all the primitives from ASIC library need to be remapped for getting the FPGA equivalent design.
4. **Pin multiplexing:** High-end FPGA may have 1000–1500 pins, and if single FPGA is sufficient to prototype the design, then the prototype has few challenges. But if IO pins required more than the pins available in single FPGA, then the real issue is due to FPGA interfaces and connectivity. The issue can be resolved by using the partitioning with the signal multiplexing. This will ensure the efficient design partitioning and efficient design prototype. This is discussed in Chaps. 14 and 15 with more details.
5. **Use of global clock sources:** Implementation of single clock domain design prototype is easy using FPGAs. But if the design has more than one clock, that is, multiple clock domains then it is quite difficult to use the clock gating and other clock generation techniques during prototype. So the migration of ASIC design into FPGA needs much more efforts and sophisticated solutions. One of the efficient solutions is to convert the larger designs into smaller design units clocked by the global clock source.
6. **Use of memories and memory models:** The memory models used in the FPGA are different as compared to ASIC. So it is essential to use the proper strategy during memory mapping. Most of the time the synthesized memory models are not available. Under such a scenario, the best possible solution is to use the prototyping board with the requirement specific memory device.
7. **Full functional test:** The full functional testing and debugging are one of the main challenges in the ASIC prototyping. During this phase, it is essential to use the debugging platform which can give the visibility of the results like speed and functional testing results.

The ASIC prototyping is achieved by using industry standard leading tools like Design Compiler FPGA. The Design Compiler is industry's leading EDA tool used to get best optimal synthesis result and best timing for the FPGA synthesis. The basic flow for the ASIC prototyping is shown in Fig. 11.1, and in the subsequent chapters, we will discuss the multiple FPGA design to get the efficient prototype.

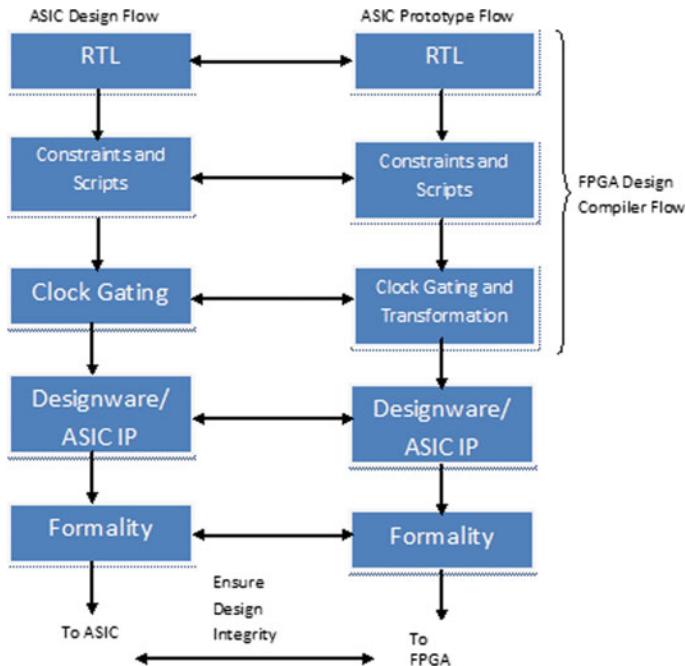
## 11.2 High-Density FPGA and Prototyping

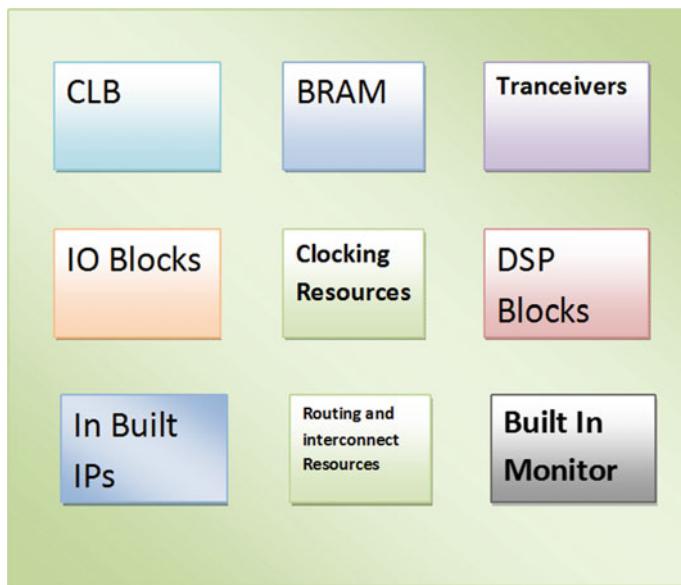
Xilinx and Intel FPGA are the market leaders in the SRAM-based FPGA. Xilinx has almost around 50–53% market share, and share of the Intel FPGA is around

**Table 11.2** Xilinx and Intel high-end FPGA [1, 2]

Technology (nm)	Low end	Mid range	High performance
120/150			Virtex II
90	Spartan 3		Virtex-4
65			Virtex-5
40/45	Spartan 6		Virtex-6
28	Artix-7	Kintex-7	Virtex-7
20/16		Kintex UltraScale	Virtex UltraScale
130	Cyclone		Stratix
90	Cyclone II		Stratix II
65	Cyclone III	Arria I	Stratix III
40	Cyclone IV	Arria II	Stratix IV
28	Cyclone V	Arria V	Stratix V
20/14		Arria 10	Stratix 10

33–35%. They also offer the one-time programmable (OTP) and nonvolatile devices. Table 11.2 gives information about the devices offered by Xilinx and Intel FPGA.

**Fig. 11.1** ASIC prototype flow [3]



**Fig. 11.2** Xilinx 7 series FPGAs

Nowadays, the FPGA devices are available with 14–16 nm technology and they are suitable for the complex SOC design emulation and prototyping. Zynq, Kintex, Virtex are few of the devices from Xilinx and can be used for the high-performance prototyping.

Intel FPGA also offers the FPGA with the process node 14 nm, and they have high performance. Few of the high-performance Intel FPGAs are Arria 10 and Stratix 10.

### 11.3 Xilinx 7 Series FPGA

Xilinx 7 series FPGA has the technology node of 28 nm and unified architecture with the voltage levels of 1 V. The FPGA architecture is scalable with common logic building blocks like CLBs, IOBs, transceivers, DSPs, PCIe, ADCs, and Clock Management Tiles (CMT). If we consider the high-end Virtex family architecture, then the architecture has the 3D multidie stacked silicon interconnects (SSI), up to 1200 user pin IOs, up to 8 Mbyte RAM, up to 2 M logic cells, and 2.4 M flip-flops. The mid-range series of this architecture is Kintex and low end is Artix.

For more information, please visit [www.xilinx.com](http://www.xilinx.com) for the architecture and package information of these FPGAs. Figure 11.2 gives information about the presence of different functional blocks in Xilinx 7 series architecture (Table 11.3).

**Table 11.3** Xilinx 7 series resources

Max. capability	Spartan-7	Artix-7	Kintex-7	Virtex-7
Logic cell	102K	215K	478K	1955K
Block RAM <sup>a</sup>	4.2 Mb	13 Mb	34 Mb	68 Mb
DSP slices	160	740	1,920	3,600
DSP performance <sup>b</sup>	176 GMAC/s	929 GMAC/s	2845 GMAC/s	5335 GMAC/s
Transceivers	–	16	32	96
Transceiver speed	–	6.6 Gb/s	12.5 Gb/s	28.05 Gb/s
Serial bandwidth	–	211 Gb/s	800 Gb/s	2794 Gb/s
PCIe interface	–	x4 Gen 2	x8 Gen 2	x4 Gen 3
Memory interface	800 Mb/s	1,066 Mb/s	1,866 Mb/s	1,866 Mb/s
VO pins	400	500	500	1,200
VO voltage	1.2–3.3 V	1.2–3.3 V	1.2–3.3 V	1.2–3.3 V
Package options	Low cost, wire bond	Low cost, wire bond, lidless flip-chip	Lidless flip-chip and high-performance flip-chip	Highest performance flip-chip

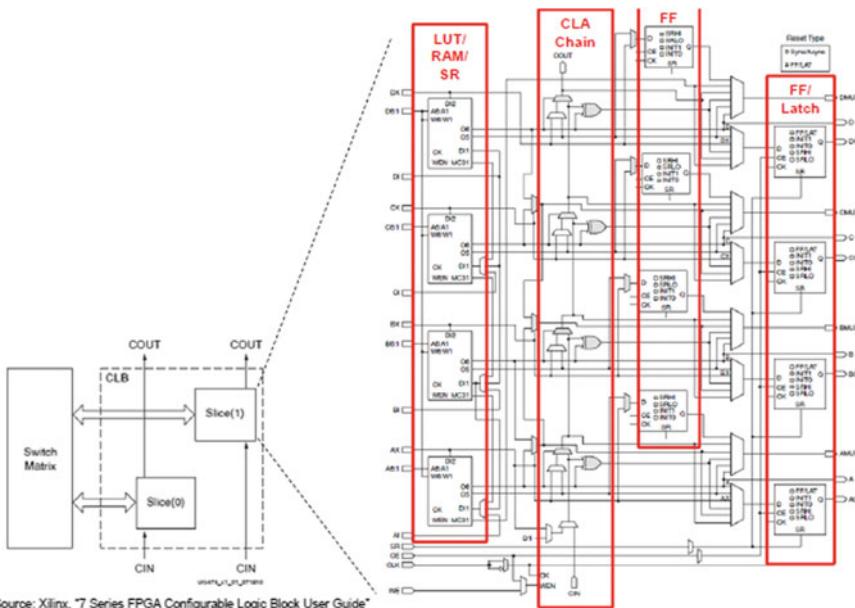
<sup>a</sup>Additional memory available in the form of distributed RAM

<sup>b</sup>Peak DSP performance numbers are based on symmetrical filter implementation

Important features of the Xilinx 7 series FPGA are shown in Fig. 11.2, and the key resources are CLB, BRAMs, DSP blocks, clocking resources, IO blocks, in-built IPs, routing and interconnect resources, transceiver and **system monitor** features. This section discusses them with few considerations which can be useful during the prototyping.

### 11.3.1 Xilinx 7 Series CLB Architecture

The Xilinx 7 series CLB is shown in Fig. 11.3, and as shown, it has two slices per CLB and each CLB consists of six input LUTs; two slices are named as SLICEM and SLICEL: SLICEM consists of RAM/SR, and SLICEL consists of only logic. As shown, it has four flip-flops or latches and can be configured depending on the bitstream of the functionality. The CLB architecture has wide multiplexers and carry chain logic. So the architecture of CLB is efficient enough to implement six inputs and multiple output combinational or sequential logic.



Source: Xilinx, "7 Series FPGA Configurable Logic Block User Guide"

**Fig. 11.3** Xilinx 7 series CLB architecture [2]

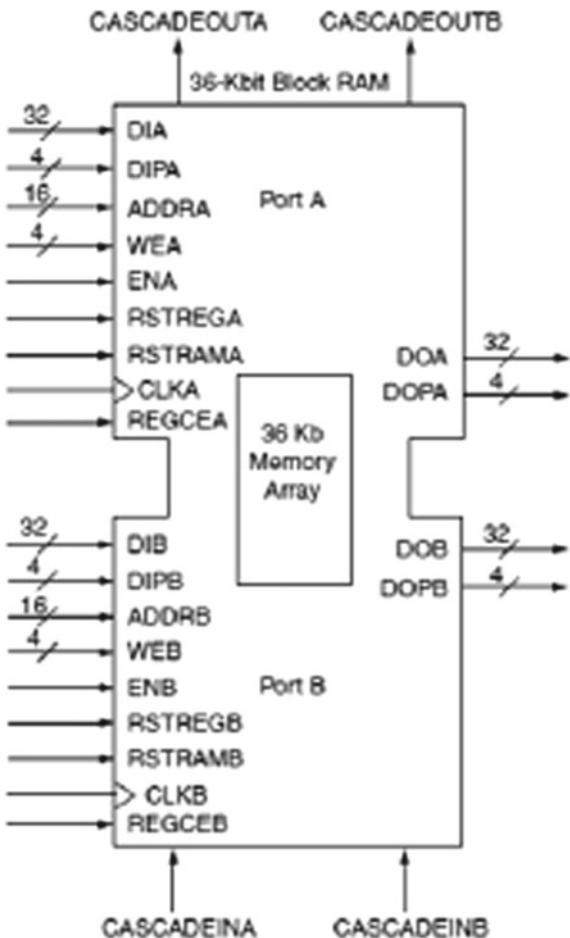
### 11.3.2 Xilinx 7 Series Block RAM

As discussed in Chap. 9, the BRAMs are used in many applications and the architecture of BRAM is vendor dependent. They can be configured using vendor-dependent EDA tool for the required capacity. The Xilinx 7 series architecture has 36 KB BRAM which can be visualized as  $2 \times 18$  KB BRAM. The BRAM is synchronous RAM and can be cascaded without any logic overheads to get  $64\text{K} \times 1$ . The BRAM can be used as single port and dual port. In the dual-port mode, the 18 KB BRAM can be used and configured as  $18\text{K} \times 1$ ,  $9\text{K} \times 2$ ,  $8\text{K} \times 4$ ,  $4\text{K} \times 9$ , etc., and 36 KB BRAM can be used as  $1\text{K} \times 36$ ,  $2\text{K} \times 9$ ,  $4\text{K} \times 9$ , etc. The BRAM architecture has built-in error correction (64-bit ECC), and they can be used also in FIFO mode (Fig. 11.4).

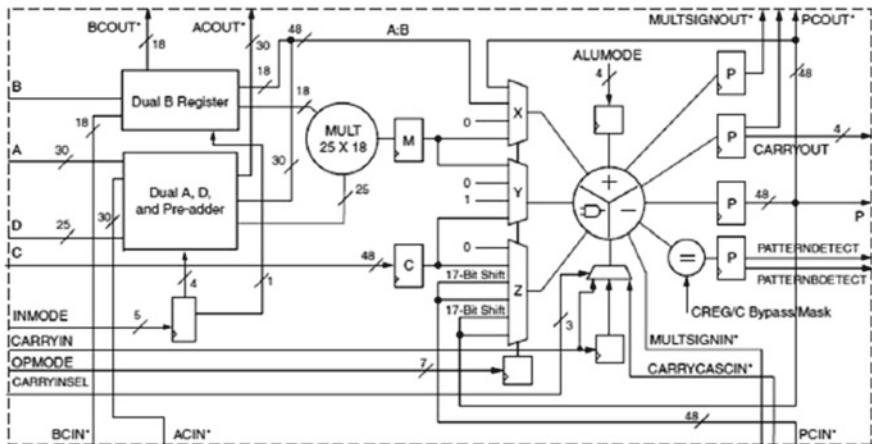
All kinds of the SOC design use the memories of type RAM, ROM, content addressable. So let us think that how these kinds of memories can be implemented. Either the memories can be instantiated from the cell library or from memory generator.

To implement the small memories of few bits, the LUTs can be used. It is important that these memories can be efficient enough to load, store, and pass the data. But to have the better and efficient architecture for the design instead of distributing the memories over the FPGA fabric, it is always better to use the BRAMs. The main features of BRAM are:

**Fig. 11.4** Xilinx 7 series  
BRAM [2]



1. **Synchronous memory:** BRAMs can implement the synchronous single- or dual-port memory. One of the real beauties of such memory blocks is that when configured as dual-port RAM each port can operate at different clock frequencies.
2. **They can be configured:** The BRAM block is dedicated dual-port synchronous RAM block and can be configured as discussed above. Each port can be configured independently.
3. **BRAMs and their use in the FIFO designs:** The BRAMs can be used to store the data and as they are dedicated and configured. With additional logic FIFOs can be implemented using BRAMs. The depth of the FIFOs can be configured with the restriction that both the read and write side should have same width.
4. **Error correction:** Consider the BRAM is configured as the 64-bit RAM, then each BRAM can store additional Hamming code bits. These bits are used to perform the single-bit and double-bit error corrections during the read process.



**Fig. 11.5** Xilinx 7 series DSP slice [2]

For 64-bit BRAM, each BRAM can store the 8-bit Hamming code. The error correction logic can be also used while writing or reading from the external memories.

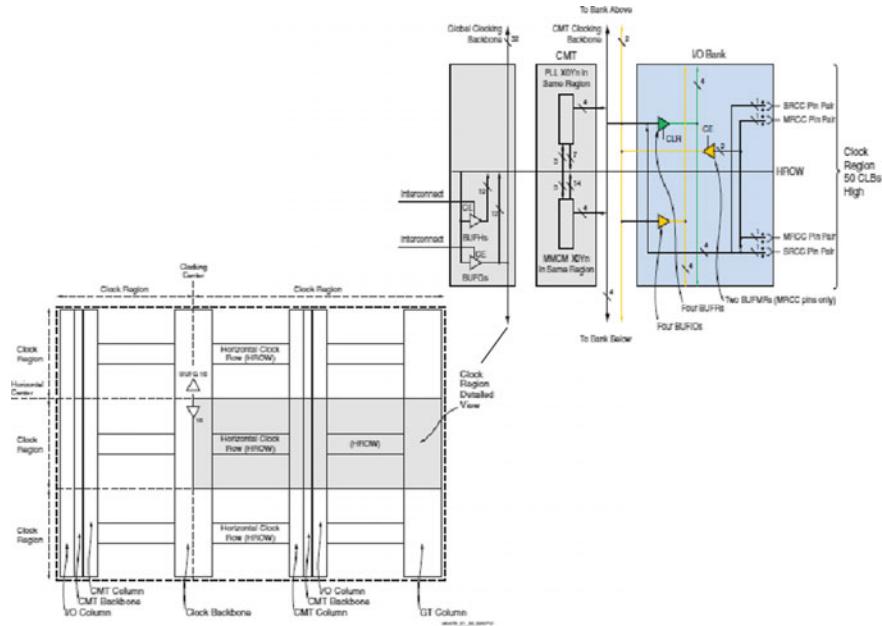
#### So let us think how the BRAMs are inferred?

The synthesis tool partitions the larger memories into small blocks, and each block can be implemented using BRAM. Effectively in the simple words, the BRAMs are very effective building block which is inferred automatically during synthesis and they are combined to model the wide range of memories used for the SOC.

#### 11.3.3 Xilinx 7 Series DSP

The Xilinx 7 series DSP48E1 slice architecture is shown in Fig. 11.5, and it supports 96-bit multiply accumulate (MACC) operation. DSP slice has 25-bit preadder, 25 X 18-bit signed multiplier, and 48-bit ALU. Also to implement the DSP algorithms, the necessity of the shifter can be accomplished by using 17-bit shifter and pattern detector.

The slice also has cascade paths to implement wide range of DSP functions and algorithms. It also provides the implementation of pipelined logic for the 12/24 bit of the data.



**Fig. 11.6** Xilinx 7 series clocking [2]

### 11.3.4 Xilinx 7 Series Clocking

For any design, the performance is dependent on the clock generation logic and architecture used for clocking the multiple CLBs. The clocking architecture for the Xilinx 7 series devices is shown in Fig. 11.6 [2], and as shown, the clock is divided into the clock regions, and effectively, it is half-width for every 50 CLBs. If we try to perceive the architecture closely, then we can conclude that every clock region can be treated as group of 20 DSPs, 10 BRAMs, 4 transceivers, 50 IOBs, and PCIe.

As shown in Fig. 11.6, it has CMTs and the tiles are used adjacent to IO columns, that is, one tile per region and two columns per device. It has one center clock spine, that is, 1 horizontal clock row/region.

### 11.3.5 Xilinx 7 Series IO

As discussed in the earlier section, the overall performance of the SOC is dependent on the available IOs and their bandwidth while exchanging the data. The high-range IOBs support standard up to 3.3 V, and the high-performance IOB supports the standard up to 1.8 V. The Xilinx 7 series has various types of IOs, and they are listed in Table 11.4.

### 11.3.6 Xilinx 7 Series Transceivers

The architecture has the low-power gigabit transceiver. Due to low-power architecture, the chip-to-chip interface is optimized and this is one of the powerful features of this FPGA. The high-performance transceiver is capable to support the data rate from 6.6 to 28.05 Gb/s depending on the family of the Virtex-7 FPGA.

The transceiver count is 16 in the Artix-7 FPGA family, up to 32 transceivers in the Kintex-7 family and up to 96 transceivers in the Virtex-7 family.

To improve the IP portability, the architecture of the serial transceiver uses the ring oscillators and LC tank circuit. The transmitter and receiver circuits are different, and they use the PLL to multiply the reference clock by the programmable number up to 100 to get the bit serial clock.

#### 11.3.6.1 Transmitter

**The following are the key features of gigabit transmitter:**

1. The transmitter is parallel to serial converter with conversion ratio of 16, 20, 32, 40, 64, or 80.
2. The GTZ transmitter supports up to 160-bit data width.
3. It uses TXOUTCLK used to register the parallel data.
4. The incoming parallel data are fed through an optional FIFO, and to provide the sufficient number of transitions, it has additional support of 8 B/10 B, 64 B/66 B encoding schemes.
5. The output of these transmitters drives the PC board with the single-channel differential output signal.
6. To compensate for the PC board losses, the output signal pair has programmable signal swing.
7. To reduce the power consumption, this swing can be reduced for the shorter channels.

**Table 11.4** Xilinx 7 series IOs

S. No	IO	Description
1	Input–output block (IOB)	The architecture has 2 columns per device and 50 IOBs per bank. Two distinct IOB types are high range and high performance
2	High-speed serial IO transceiver	Transceivers are available in quad that is four per block. They have different types and multiple standards: GTH/GTP/GTX/GTZ with the bandwidth of 3.75–28 Gbps
3	PCI express (PCIe) blocks	They are build on GTX serial IO transceivers, and they are compatible with the Gen 1, Gen 2, and Gen 3 protocols having 2.5, 5 and 8 Gbps bandwidth
4	XADCs	For every analog to digital converter, there is analog sensor and the architecture has 12-bit two ADCs

### 11.3.6.2 Receiver

The following are the key features of gigabit receivers:

1. The receiver is serial to parallel converter with conversion ratio of 16, 20, 32, 40, 64, or 80.
2. The GTZ receiver supports up to 160-bit data width.
3. To guarantee sufficient data transition, it uses non-return-to-zero (NRZ) encoding.
4. The parallel data are transferred to the FPGA using the RXUSRCLK
5. For short channels to reduce power consumption by almost 30%, the transceiver offers special low-power (LPM) mode.

### 11.3.6.3 In-Built IPs

Nowadays, FPGAs are more capable to be used in the communication, medical imaging, and networking areas due to available PCIe, Ethernet MAC, Phy, and processor cores on the FPGA fabric.

The Xilinx Virtex-7 series FPGA has the PCIe, Ethernet MAC, Phy integrated on the FPGA fabric.

- **Ethernet:** It can operate with the speed of 2.5 Gbits/s, and the Ethernet block is designed with the IEEE standard 802.3-2005
  - The four tri-mode (10/100/1000 Mb/s) MAC blocks can be connected to the FPGA fabric and transceivers.
- **PCIe:** The FPGA has the integrated interface block for the PCI express and can be configured as the endpoint. It can operate at the speed of the 2.5 Gbit/s and 5 Gbit/s data rate.
- **CPU hard core:** The optimized ARM IP core can run at the higher speed (10 times more) as compared to RTL, and it can be used during prototyping if these SOC processor core features are matching with the available IP core. If multiple processors are required during the prototyping, then use the multiple FPGAs with the partitioned logic but this may reduce the performance of the prototype due to the bus access time. These blocks are not directly inferred by synthesis tool but need to be instantiated using core generators.

### 11.3.7 Built-in Monitor

It can be used during prototyping to get the thermal and power supply information, and it does not need any instantiation in the design. Once the power connections are made, then using the Test Access Port (TAP) the data can be accessed at any time.

## 11.4 Important Takeaways and Further Discussions

1. The moderate gate count FPGA consists of the CLBs, BRAM, routing and inter-connect resources, DSP slices, IO blocks, and clock management network.
2. The FPGA architecture is vendor specific, and the programming can be achieved by using vendor-specific EDA tool chain.
3. FPGAs are used extensively during this decade for the prototyping and for the emulation.
4. The emulation using FPGA can be cost-effective and efficient way to test the functionality to achieve the desired performance.
5. The high-end FPGAs from Xilinx and Intel can be used to validate the SOC design, and these FPGAs consist of the processor core which operates on higher clock frequency.
6. For SOC design and prototyping, the hardware and software partitioning can play important role and the overhead of the communication between the hardware and software can be reduced by using the pipelining and multitasking.
7. The IO interface bandwidth and multitasking features need to be incorporated into the design to achieve the required design performance.
8. The hard processor IP cores can be used during prototyping if the SOC processor core feature matches with the available IP core.

This chapter has given understanding about the FPGA functional blocks. Now the main question is that whether FPGA synthesis and ASIC synthesis yield to the same result. Answer at high level is no! The reason being for ASIC the netlist consists of the standard cells and macros. For the FPGA synthesis, the logic is inferred using CLBs (LUTs, registers, and other cascade logic), BRAMs, DSP blocks, transceivers, and other FPGA vendor-specific blocks.

The next chapter focuses on the SOC prototyping guidelines. While SOC prototyping we need to have modification in the ASIC/SOC RTL (clock gating logic, clock enable logic, memory blocks, and IO pads), and it is discussed with the practical considerations in the next chapter.

## References

1. [www.altera.com](http://www.altera.com)
2. [www.xilinx.com](http://www.xilinx.com)
3. [www.synopsys.com](http://www.synopsys.com)

# Chapter 12

## SOC Prototyping Guidelines



*Partition the design into multiple FPGAs using the automatic partitioning. The partitioning tool can give the better partitioning results.*

**Abstract** The chapter discusses important design guidelines used during the SOC prototyping. The prototyping performance is based on how the design is partitioned into multiple FPGAs? What is IO speed and bandwidth? And how synchronizers are used? The chapter focuses on all these aspects in much more detail with the practical examples and considerations. Although most of the guidelines are discussed in the previous few chapters, in this chapter they are documented to have better understanding and their use during SOC prototyping.

**Keywords** Partitioning · Register IO · Combinational loop · Oscillatory behavior  
Combinational output · Unintentional latches · Synchronous designs  
Asynchronous designs · Multiple clock domain designs · Clock gating  
Multiple FPGA partitioning · TDM · LVDS · SERDES  
Combinational boundaries · Sequential boundaries

As discussed in the previous few chapters, the prototype engineer needs to consider many aspects to achieve the better performance for the SOC prototype. If we try to understand this in depth by considering the speed of the SOC, logic complexity, multiple clock domains and multiple power domain designs, then we can conclude that the million gate SOC prototype cannot be realized using the single FPGA. So the design needs to be partitioned into multiple FPGAs. The chapter discusses the SOC design guidelines and recommendation to have the better FPGA platform.

## 12.1 What Guidelines I Should Follow During SOC Prototyping?

What I should think about the efficient prototyping platform to validate the SOC? This is the first fundamental question which needs to be answered! The answer is the platform should have the efficient architecture and the better test and debug plan to achieve the desired performance for the SOC! The following section gives information about the same:

1. **Allocate the primary and secondary responsibilities:** It is very much required to allocate the primary and secondary responsibilities to the team members working for the prototype. It is always advisable to have the communication between the RTL design team and the SOC prototype team. This will yield into the better result, and both teams will be able to understand the risk during the design and prototype.
2. **Comparative milestones:** It is advisable to compare the result of the SOC prototype with the golden reference. This comparative method is useful to understand the stage of the prototype. For example, the functional converge during verification is 97%, and at the prototype level, it is almost 80%. If this information is captured, then the improvements can be made in the existing design/test and debug environment to achieve the higher coverage at the board level also.
3. **Version control:** Use the version control for the software, and the RTL modification as the database is useful during the SOC prototype cycle.
4. **List of deliverables and milestones:** Create the database of the deliverables or milestones for the prototype. Following can be the deliverables for the SOC design
  - System requirement specifications
  - Risk and dependability document
  - C/C++ functional golden reference design
  - Architecture of the SOC
  - Micro-architecture for the SOC
  - Clocking network and distribution
  - Pin count of the SOC and their functionality
  - RTL design versions
  - RTL verification plan and versions
  - SOC synthesis and timing scripts with the design constraints
  - Timing, area, and power reports for the individual functional blocks
  - Top-level area, timing, and power constraints and the reports
  - Implementation constraints and target FPGA architectures
  - Multiple FPGA partitioning and board layouts
  - Test and debug plans
  - Technology-independent design: Have a practice to use the technology-independent design. To preserve the RTL, use the ‘define’ and ‘ifdef’ macros

## 12.2 RTL Modifications to Have FPGA Equivalent

There are multiple scenarios in which the RTL modifications are required during the SOC prototype. Few of them have been listed in this section:

- Gated clock instantiation:** The gated clock structure for the SOC may not be matched with the FPGA equivalent structure, and hence, it is essential to modify the RTL to infer the gated clock structure (Figs. 12.1 and 12.2).
- SOC IPs:** Most of the IPs the RTL is not available, and hence, it is essential to have the FPGA equivalent of such IPs.
- ASIC/SOC memories:** The memory structure for the ASIC or SOC is not identical with the FPGA memories, and hence, it requires the modification during the prototype stage.
- Top-level pads:** AS FPGA tool does not understand about the instantiation of the pad, and hence, it is essential to modify them during the prototype. As it does not handle the IO PAD in the RTL and infers the FPGA PAD. So need to leave the pads out with dangling connections inactive or to the top-level boundary. For the prototype, replace each IO pad instance with synthesizable model of FPGA equivalent.

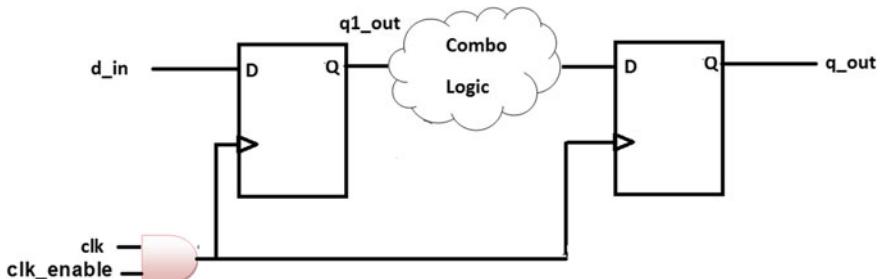


Fig. 12.1 Gated clock used for the design

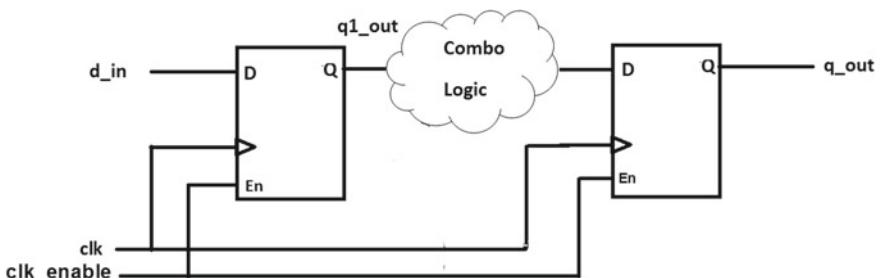
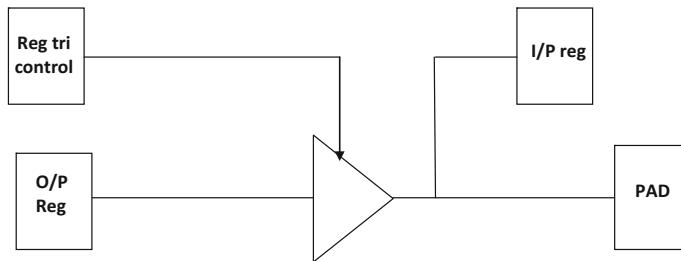


Fig. 12.2 FPGA equivalent clock gating



**Fig. 12.3** FPGA basic IO cell

The model should have the logical connections at the RTL level and that can be done by writing small piece of code using the Verilog RTL. For the efficient prototype, prepare the SOC pad library. The basic FPGA IO cell is shown in Fig. 12.3.

5. **IPs in the netlist forms:** The netlist form may not be the FPGA equivalent and hence needs modification during prototype.
6. **Leaf cells:** Leaf cells from the ASIC library may not be understood by the FPGA, and hence, it needs modifications.
7. **Test circuitry:** The Built-In Self-Test (BIST) and other test or debug circuit need to have the FPGA equivalent and hence needs the modification.
8. **Unused inputs:** For the unused input pins, it is essential to tweak the RTL.
9. **Generated clocks:** During prototype to achieve the better performance, the generated clocks need to be modified by its FPGA equivalent.

## 12.3 What Care I Should Take During Prototyping?

### 12.3.1 Avoid Use of Latches

Although the latch-based design is better to save the power, it is advisable to use flip-flop based design. The flip-flop based design can guarantee the clean timing paths.

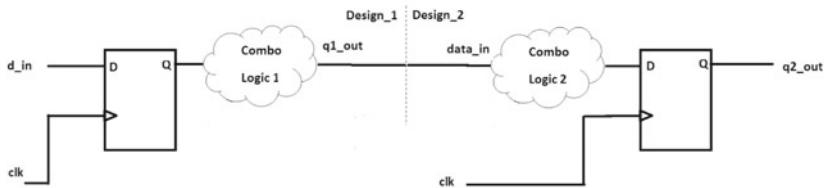
### 12.3.2 Avoid Longer Combinational Paths

As inside the FPGA, combinational logic is mapped using the LUTs, and it is recommended to avoid the longer combinational paths. Although the LUT delay is uniform, the longer combinational paths degrade the performance of the SOC prototype. It is advisable to break the long combinational paths into the shorter paths by using the pipelined registers. Although there is additional overhead on the area due to use

**Practical Scenario:** If the output of the design is not registered then it has impact on the other module. In the design the output is combinational and given as input to other module then it will increase the combinational delay in the design. As the data path delay has increased it affects on the timing and may violate the setup time. To avoid this register all the inputs and outputs.

**FPGA Synthesis:** In the FPGA design the register to register path is between the CLB flip-flops. Every CLB has flip flops and the LUTs.

**ASIC Synthesis:** In the ASIC the standard cells are used and the register to register path is between the flip-flops. The standard cell library has the logic gates and the flip-flops. Consider the following design where the output is from combinational logic that is output is not registered.



If  $q1_{out}$  drives other design block then as stated above it increases the delay in the data path. Let us consider the design scenario shown in above two figures.

The output from the first design is  $q1_{out}$  and given to  $data\_in$  input of the second design. This will reduce speed of the design. The data arrival time (AT) =  $T_{pdff} + t_{combo1} + t_{combo2}$  and the data required time (RT) =  $T_{clk} - tsu$ . So for positive slack  $RT \geq AT$  and the period of clock is  $T_{clk} = T_{pdff} + t_{combo1} + t_{combo2} + tsu$ .

If the delay of flip-flop ( $t_{pdff}$ ) = 1 ns, delay of combo logic 1 ( $t_{combo1}$ ) = 0.5 ns, delay of combo logic 2 ( $t_{combo2}$ ) = 0.5 ns, setup time ( $tsu$ ) = 0.5 ns and hold time ( $th$ ) = 0.25 ns then the clock time Period ( $T_{clk}$ ) =  $T_{pdff} + t_{combo1} + t_{combo2} + tsu = 2.5$  ns and maximum operating frequency is 400 MHz.

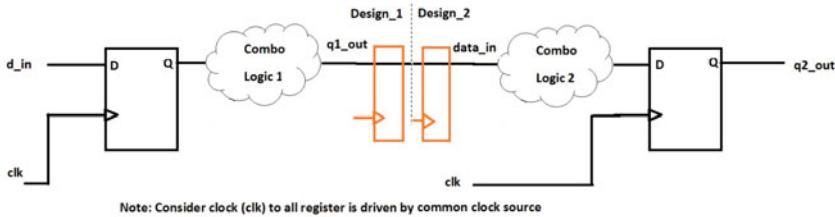
If inputs and outputs of all the designs are registered then the design has clean register path. Even it improves the performance of the design.

The improved design is shown below and it improves the register to register path timing.

### Practical Scenario 12.1 The impact of registered output

of the pipelined register, the shorter path design is best placed, mapped, and routed using the adjacent CLB resources. The practical scenario is described in Example (Practical Scenario 12.1 and 12.2).

**Registered Inputs and Outputs:** As the inputs and outputs of the design are registered it has impact on the performance of the design. The data path delay is improved by this technique.



The clock time period ( $T_{clk}$ ) is  $T_{clk} = T_{pdff} + t_{combo1} + t_{su}$  for the first design and for the delay of the second design ( $T_{clk}$ ) is  $T_{clk} = T_{pdff} + t_{combo1} + t_{su}$ . If the delay of flip-flop ( $t_{pdff}$ ) = 1 ns, delay of combo logic 1 ( $t_{combo1}$ ) = 0.5 ns, delay of combo logic 2 ( $t_{combo2}$ ) = 0.5 ns, setup time ( $t_{su}$ ) = 0.5 ns and hold time ( $t_{th}$ ) = 0.25 ns then the clock time Period ( $T_{clk}$ ) =  $T_{pdff} + t_{combo1} + t_{su} = 2.0$  ns and maximum operating frequency is 500 MHz.

Thus the performance of the design has improved. Previously without the registered inputs and registered output the frequency was 400 MHz and with the registered inputs and outputs the frequency is 500 MHz.

### Practical Scenario 12.2 Registered inputs and outputs

#### 12.3.3 Avoid the Combinational Loops

To avoid the oscillatory behavior, it is recommended to avoid the combinational loops. The oscillatory behavior is unpredictable, and it may be due to the missing signals in the sensitivity list, incomplete **case** statements, incomplete nested **if** statements.

#### 12.3.4 Use Wrappers

Most of the FPGA vendor supports the RTL description in the generic logic form or in the Synopsys DesignWare component form. So do not use the technology-dependent cells. At the leaf level, introduce the technology-specific details.

It is mandatory to take care that the source changes should have the lesser impact on the design. For this use the wrappers and make changes inside the design elements. If the SOC architecture uses RAM, then make changes inside the library elements rather than in the RTL. This improves overall portability of the design.

### ***12.3.5 Memory Modeling***

If we consider the SOC design, then the memory is technology specific and may not be FPGA equivalent. Under such circumstances, use the FPGA compatible memory versions during prototype. As stated earlier, use the wrappers around the technology-dependent elements in the design. This should be done for the macros, RAMs in the technology library.

### ***12.3.6 Use of Core Generators***

Use the Xilinx core generators and specify the target technology, core type, and initialization sequence. The FPGA tool is used to generate the netlist and initialization files. Place and route tool uses the netlist to perform the placement of the functional block on the FPGA fabric. Initialization can be accomplished by the template files. Instantiate the template files in the design. Most of the time we observe that the tool-generated wrapper file consists of the functional stimulus data and can be used for functional simulation of the design.

### ***12.3.7 Formal Verification***

The big question which needs to address is that how I can verify the functionality of FPGA and SOC RAM? Whether they have the same behavior or not? The answer is simple; it can be done by using formal verification (FV) during the early stage of prototyping. This can verify the equivalence checking for the FPGA and SOC RAM!

### ***12.3.8 Blocks Not Mapping on the FPGA***

The analog blocks, IPs, may not be directly mapped on the FPGA as the RTL code/netlist of these blocks is not available. Under such circumstance, use the evaluation boards supplied by the IP vendor or design the functional equivalent FPGA for the IPs and interface them with the FPGA.

### ***12.3.9 Better Architecture Design***

It is always better practice to have the efficient architecture and micro-architecture for the SOC and FPGA designs. The architecture document should have the information about the tweaking or modification required for the FPGA prototype.

### ***12.3.10 Use Clock Logic at Top Level***

For fine portability and modularity, keep the clock distribution/generation logic at the top level.

### ***12.3.11 Bottom-Up Approach***

Use the bottom-up approach for the design, and this can be good to generate the constraints at the block level and chip level. One can think about keeping the high-level modules free from the use of parameters and generics.

## **12.4 SOC Prototype Guidelines for Single FPGA Design**

If the design is moderate gate count, then the design can be mapped using the single FPGA. Under such circumstances, what care we should take?

1. Partition the design to have the architecture equivalent functionality using the FPGA resources.
2. Instead of using the distributed RAM, use the BRAMs.
3. Use the DSP blocks to infer the DSP functionality.
4. Use the 60–70% of the resources of FPGA that will give the room for the designer and prototype team to add the functionality.
5. Use the logic replications and resource sharing techniques for the better performance.
6. Have a clock network in the separate block, and use the synchronizers to pass the data between the multiple clock domains.
7. Use the high-speed IOs for data transfer between the FPGA and other associated peripherals, controllers.
8. Use the TDM and LVDS for the data transfer across the FPGA boundaries.

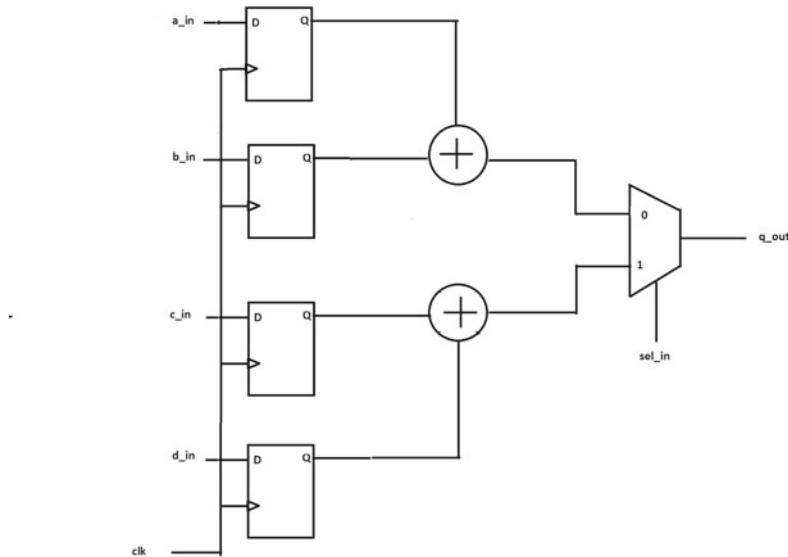
Few Design scenarios are listed below

**Practical Scenario:** Most of the time during the design we observe the use of the same resources multiple times. This can increase the area.

**FPGA Synthesis:** In the FPGA design the design uses the LUTs and registers. CLB consists of multiple LUTs and registers and the CLB architecture is vendor dependent.

**ASIC Synthesis:** In the ASIC the standard cells are used. The standard cell library has the logic gates and the flip-flops.

Consider the following design where the output is from combinational logic that is output is not registered.



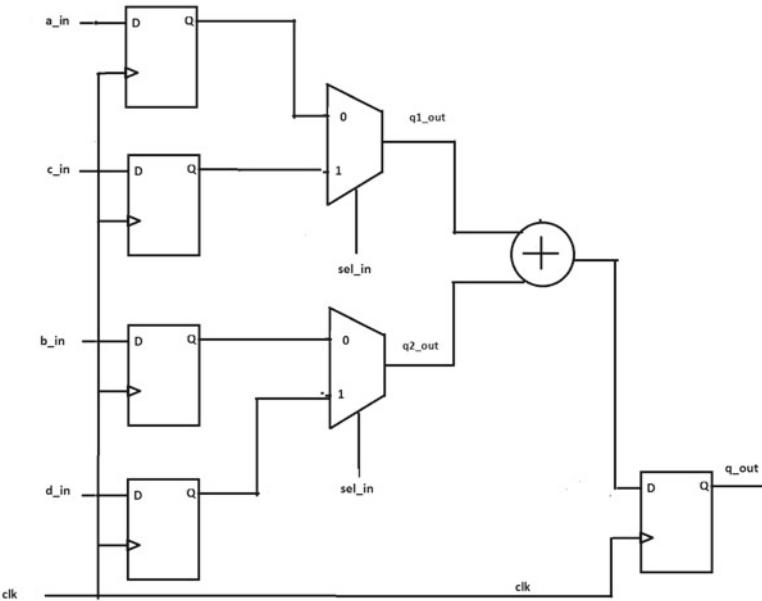
As shown in above figure, for the sel\_in is equal to logic '1' the  $q_{out} = c_{in} + d_{in}$  and for the sel\_in is equal to logic zero the  $q_{out} = a_{in} + b_{in}$ . In this the logic infers two adders and single multiplexer. The disadvantage is both adders perform the addition simultaneously and  $q_{out}$  is dependent on the status of sel\_in input of 2:1 Mux. This unnecessary increase the area and even this is not the real requirement. Even this kind of module has unregistered output so if  $q_{out}$  is used as input to other module then unnecessary this can increase input insertion delay.

**Practical Scenario 12.3** The design without resource sharing

### 12.4.1 Practical Scenarios and Use of Resources

See Practical Scenarios [12.3](#) and [12.4](#)

To improve the performance of the design let us perform only one operation at a time using more number of multiplexers and less number of adders. This technique is called as resource sharing and can be efficient way to improve the performance of the area.



As shown in the above figure the adder is used as common resource and chain of multiplexer is used to pass the inputs to the adder. For the status of select input  $sel\_in$  is equal to 1 the output is  $c\_in + d\_in$  as the first input line of both multiplexer is selected.

For the status of select input  $sel\_in$  is equal to zero the 0th input of multiplexer is selected and  $a\_in, b\_in$  is passed to  $q1\_out$  and  $q2\_out$  respectively. For  $sel\_in$  is equal to zero the  $q\_out = a\_in + b\_in$ .

Due to use of more number of Mux and less number of adders the performance of the design is improved. If we consider the input width as 32 bit then this technique can save lot of area.

#### Practical Scenario 12.4 Performance improvement using resource sharing

##### 12.4.2 Efficient Use of FPGA Resources

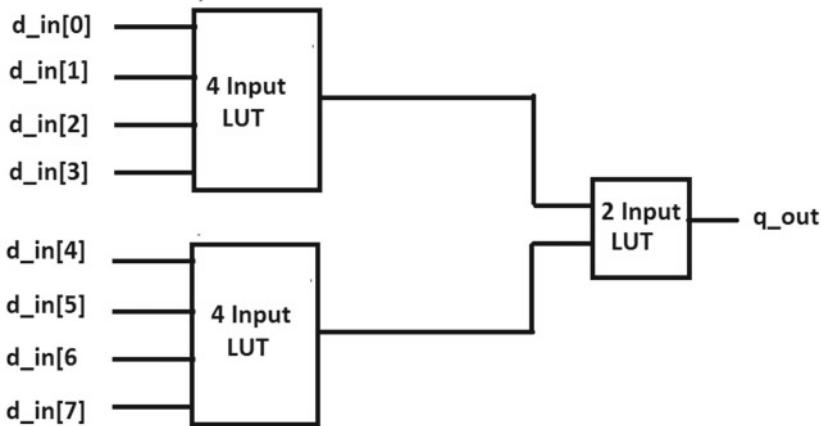
See Practical Scenario [12.5](#)

##### 12.4.3 Use of Multiple LUTs in the FPGA Design

See Practical Scenario [12.6](#) and [12.7](#)

**Practical Scenario:** Purely Combinational path in the design

**FPGA Synthesis:** The FPGA consists of number of CLBs and the CLB architecture is vendor dependent. Each CLB consists of few LUTs and few registers. Consider the logic inferred for the “ $q_{out} = (d_{in}[0] \& d_{in}[1] \& d_{in}[2] \& d_{in}[3]) | (d_{in}[4] \& d_{in}[5] \& d_{in}[6] \& d_{in}[7])$ ”. The logic uses two four input LUTs and single 2 input LUT



As shown in above figure, the design has combinational path

1. Input to Output path (Combinational Path): From the above figure it is clear that the path from input to output ( $d_{in}[7:0]$  to  $q_{out}$ ) is input to output path or also called as combinational path.

For any designer it is essential to have information about CLB architecture of the FPGA family. CLB architecture is vendor dependent and consists of multiple input LUTs, registers, carry chain etc.

**Practical Scenario 12.5** LUT uses to implement the combinational design

## 12.5 Prototyping Guidelines for Multiple FPGA Designs

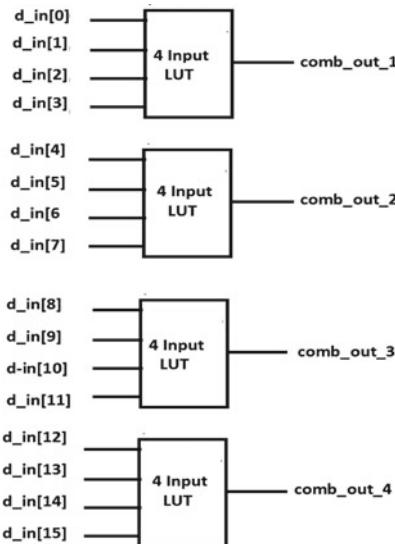
If the design has million gates, then obviously design does not fit inside single FPGA. Under such circumstances, we need to have the multiple FPGA designs. As stated earlier, the prototype performance of the multiple FPGA designs is based on how the design is partitioned. The better partitioning design can yield into the better speed and performance of the design. Following are few of the recommendations which are useful in the multiple FPGA designs

1. Have the partition of the design into analog and digital blocks.
2. Use daughter cards for the analog interface with the FPGA. That is, using the separate boards for the ADC and DAC logic

**Practical Scenario:** Logic inferred by only LUTs that is purely combinational paths in the design

**FPGA Synthesis:** The FPGA consists of number of CLBs and the CLB architecture is vendor dependent. Each CLB consists of few LUTs and few registers. Consider the logic inferred for the

```
assign comb_out_1 = d_in[0] & d_in[1] & d_in[2] & d_in[3];
assign comb_out_2 = d_in[4] | d_in[5] | d_in[6] | d_in[7];
assign comb_out_3 = d_in[8] ^ d_in[9] ^ d_in[10] ^ d_in[11];
assign comb_out_4 = !(d_in[12] ^ d_in[13] ^ d_in[14] ^ d_in[15]);
```



As shown in above figure, the design has combinational path

2. Input to Output path (Combinational Path): From the above figure it is clear that the path from input to output ( $d_{in} \{7:0\}$  to  $q_{out}$ ) is input to output path or also called as combinational path.

For any designer it is essential to have information about CLB architecture of the FPGA family. CLB architecture is vendor dependent and consists of multiple input LUTs, registers, carry chain etc.

#### Practical Scenario 12.6 Parallel logic using multiple LUTs

3. Partition the design by using the partitioning tool and estimate the resources required for the design.
4. Choose the target FPGA device and use only 60–70% of the FPGA resources.

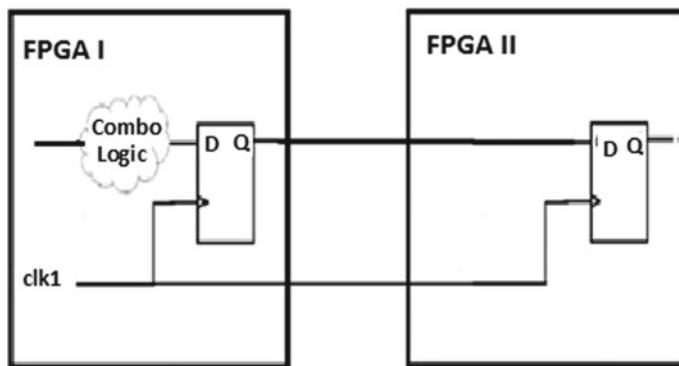
As shown in above figure, the design has combinational path

3. Input to Output path (Combinational Path): From the above figure it is clear that the path from input to output ( $d_{in} \{15:0\}$ ) to respective  $comb\_out$  is input to output path or also called as combinational path.

For any designer it is essential to have information about CLB architecture of the FPGA family. CLB architecture is vendor dependent and consists of multiple input LUTs, registers, carry chain etc.

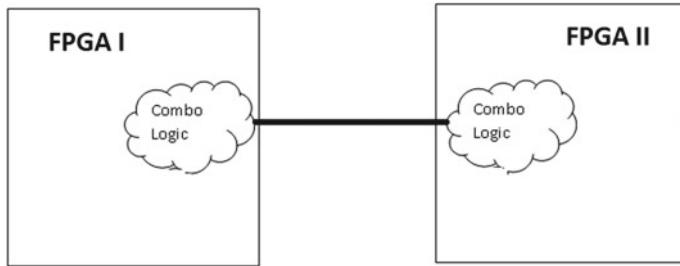
The synthesis tool infers 4 LUTs and each LUT has four inputs. The architecture of the CLB is vendor specific. Consider every CLB has four inputs and single output then the synthesis tool uses two CLBs and 4 LUTs.

#### Practical Scenario 12.7 Parallel logic using multiple LUTs (contd.)

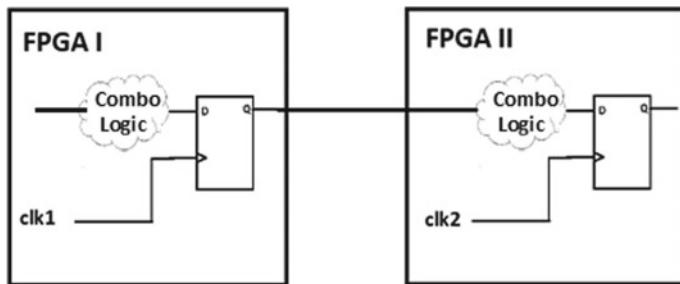


**Fig. 12.4** Multiple FPGA synchronous designs

5. Have a partitioning of the design by considering the requirement of high-speed IOs, LVDS, architecture boundaries and interfaces.
6. Use the multiple FPGAs in the required topology, and for the pin count reduction, use the TDM.
7. Use the registers available in the IO blocks to have the registered outputs and registered inputs. Consider Fig. 12.4 where the registered output from the FPGA I drives the registered input at FPGA II. Both the FPGAs are driven by the  $clk1$ . Under these circumstances, the design can be fully synchronous and results into the better performance.
8. It is not recommended to have the combinational partitioning across the FPGA boundary as it increases the delay in the critical path and reduces the performance of the design. As shown in Fig. 12.5 the design using FPGA design is not partitioned rightly as it has combinational boundaries. So the overall combinational delay between two sequential elements is the addition of the combinational delay and results into the longest critical path. Such type of scenarios needs to be avoided during the design partitioning.



**Fig. 12.5** Combinational boundaries across FPGA



**Fig. 12.6** Interface using direct connection

### 12.5.1 Interfaces and Connectivity

The interface and connectivity between the multiple FPGAs are one of the factors which limits the desired performance of the FPGA. The reason is being the interconnect delay (onboard delay). If the onboard delay is minimized, then the prototype performance can be better. Consider Fig. 12.6 in which the FPGA 1 is passing the data to FPGA 2. In such design, what we need to consider is the onboard delay due to the connectivity between two FPGAs. So the overall speed of the prototype is depending on the on-chip delay (FPGA logic delays) and onboard delay.

If we use the direct interconnect, then this delay can be maximized; if we use the connectors, then the delay can be moderated; and if we use the switch matrix to establish the connectivity, the delay can be minimized.

### 12.5.2 Clocking and Speed of the Design

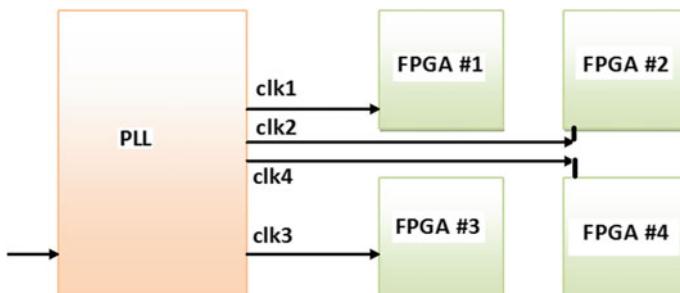
It is impossible to have the same clocking speed for the FPGAs as SOCs are faster as compared to FPGA. So it is recommended to have the FPGA prototype using the slower clock speed. If it is Virtex-7, then better FPGA prototyping can be achieved at speed of 150 MHz. The major goal is to test the functionality and the test cases passing to validate the FPGA prototype for the SOC design.

### 12.5.3 Clock Generation and Distribution

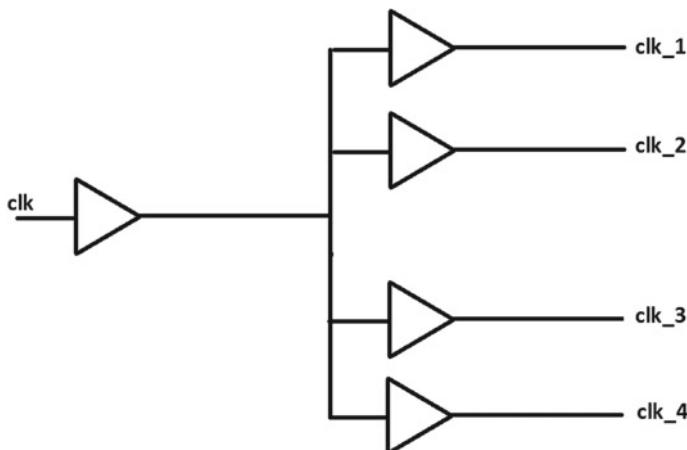
To maintain the uniform clock skew, the clock distribution logic should have uniformity in the delays. The PLL can be used to generate the clocks, and the clock network on the board should have the symmetrical routing so that the wire delays and skew can be uniform.

As shown in Fig. 12.7 the clk1 to clk4 is generated from the PLL. It can be on-board PLL, and the clocks to these FPGAs are distributed. The board designer should take care of the symmetrical distribution of the clocks for the uniform clock latency. The better clock distribution can give the better performance results for the synchronous design.

To have the clock distribution for the design having single clock, the clock tree can be created and is shown in Fig. 12.8.



**Fig. 12.7** Clock generation logic for multiple FPGAs



**Fig. 12.8** Clock tree for the uniform latency

## 12.6 IP Use Guidelines During Prototype

The IP models available from the vendors can be used for the simulation but can be used for the prototype. Hence, it is recommended to have the IP equivalent logic on the add-on board. Interface the add-on boards with the SOC prototype platform and then perform the testing for the same.

IPs are available in the following format, and the prototype team needs to use the IPs during the design cycle at the different stages.

1. **RTL Source code of IP:** In this type of IP open-source code or the license version of the IP source code is available. The source code using VHDL or Verilog can be available.
2. **Soft IP:** This type of IP cores is sometimes encrypted versions, and they need to have some processing during the design and reuse.
3. **IPs in the netlist form:** They are available in the form of the presynthesized netlist of the SOC components or Synopsys GTECH.
4. **Physical IP:** They are also called as hard IPs, and they are pre-laid out by the foundry.
5. **Encrypted Source Code:** The RTL is protected with the encrypted key and must be decrypted to get the RTL source.

If we consider the Virtex-7 device, then the AES encryption key is available for the protection, and the encryption key is 256-bit key and can be used to protect the design.

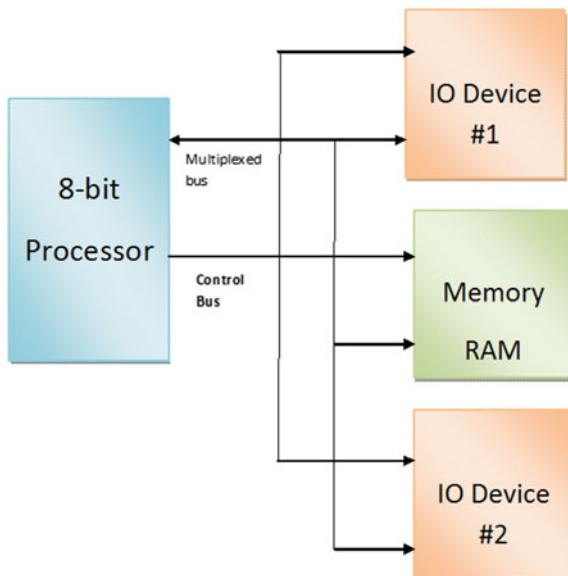
## 12.7 Guidelines for Pin Multiplexing

AS SOC has larger pin counts, and FPGA pin count may not be sufficient. To reduce the pin count that uses the pin multiplexing, consider the figure in which the processor is interfaced with the IO and memory devices. If the separate address and data bus which is used, then overall pin count is high. To reduce the pin count, use the multiplexing of the address and data bus. The 8-bit processor interfaced with the IO, and memory using multiplexed bus is shown in Fig. 12.9. Care should be taken by the prototype team to demultiplex the address and data bus while designing the decoding and selection logic.

## 12.8 IO Multiplexing and Use in Prototype

IOs can be multiplexed using the asynchronous multiplexing or synchronous multiplexing technique.

**Fig. 12.9** Multiplexed address data bus



1. **Asynchronous multiplexing:** The data transfer clock is not aligned in phase with the design/system clock.
2. **Synchronous multiplexing:** The data transfer clock is in phase with the design/system clock.

As discussed earlier, the SOC prototype is well partitioned and having the multiple FPGA designs. Still the limitation is due to available FPGA pins. Multiplexing can be used to reduce the number of pins. It is basically grouping of the identical IO signals to transfer the data serially between the FPGAs. It uses the MUX, DEMUX, transfer clock. The launch FPGA can transfer the serial data using the transfer clock, and the capture FPGA receives the serial signal at the input of DEMUX.

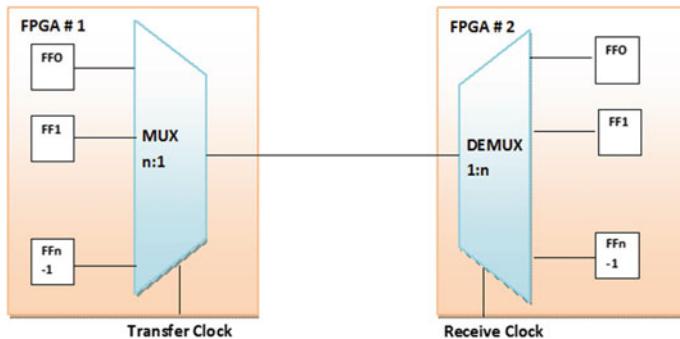
What we can do to have multiplexing is one of the important questions needs to be answered! We can have the  $n:1$  MUX logic to multiplex ' $n$ ' IO signals at the launch FPGA. The launch FPGA should use the transfer clock to transfer the IO signal information serially. The capture FPGA should have the  $1:n$  DEMUX, and it should receive the serial IO signal at the transfer clock rate.

The relationship between the transfer clock and design clock is given by

$$\text{Transfer clock} = n * \text{Design clock}$$

This should be the minimum clock value with which IO signal needs to be outputted from the launch FPGA.

For example, if we have design which is clocked at operating frequency of 25 MHz, then the minimum clock to transfer the data should be  $n * 25$  MHz. If  $n = 4$  IO signals, then the transfer clock frequency should be 100 MHz at least or it can be more than



**Fig. 12.10** IO multiplexing

100 MHz. The reason for using the faster clock to transfer the data serially is to have the availability of the data on the next active clock edge at the capture FPGA.

The figure shows the IO multiplexing using MUX and DEMUX for ‘ $n$ ’ IOs. The real issue in such kind of the design is the speed of the data transfer due to IO delays and the onboard delays.

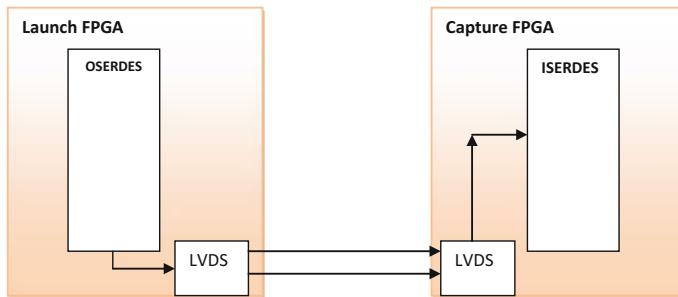
Practically, we can use the RTL tweaks to add these elements at the FPGA boundary or we can achieve the pin multiplexing after design partitioning. The EDA tool certify [1] can be used to partition the design and for IO multiplexing. Certify tool can add the CPM [1] and HSTDMD to have the multiplexing of IOs. Where CPM is certify pin multiplier [1] and HSTDMD [1] is high-speed time division multiplexing (Fig. 12.10).

## 12.9 Use of LVDS for High-Speed Serial Data Transfer

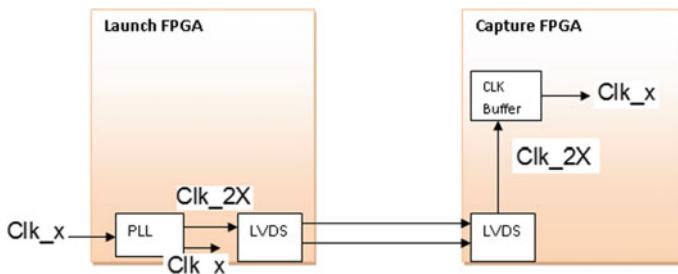
The better approach to have the pin multiplexing is by using the low-voltage differential signals (LVDS). In this strategy, the IO-SERDES can be used to pass the data on the pin. The major advantage is that it is used to pass the high-speed data on serial line (Fig. 12.11).

## 12.10 Use the LVDS to Send Clock on Parallel Line

The LVDS can be used to send the clock using the parallel lines. Figure 12.12 shows one of the mechanisms to send the clock instead of generating the clock at capture FPGA.



**Fig. 12.11** LVDS and IO SERDES for serial transfer



**Fig. 12.12** Transfer of clock using LVDS

## 12.11 Use the Incremental Flows

Use the incremental synthesis and place and route (P & R) flows during the SOC prototype to have quick turnaround time. The time duration and efforts of many weeks or days can be saved using the incremental flows enabled at the front-end and back-end design.

## 12.12 Important Takeaways and Further Discussions

As discussed in this chapter, the SOC prototype can use single or multiple FPGAs and following are few takeaways.

1. The design should be well partitioned across the FPGA boundaries.
2. Partition the design at register boundaries, i.e., registered inputs and registered outputs.
3. Use the design and IO constraints while performing the timing analysis.
4. The onboard delays should be minimum in case of multiple FPGA designs to have the better performance.
5. Use the clock gating conversions for the FPGA designs.

6. In the multiple clock domain designs, replicate the synchronizers in the FPGA.
7. Use the clock network to have the minimum balanced skew across the board.
8. Use the compatible daughter cards for the analog blocks or IP communication with the FPGA.
9. Avoid the combinational loops in the design as it results in the oscillatory behavior.
10. Use the 60–70% FPGA resources and then try to find the FPGA count to realize SOC.

The next chapter focuses on the design partitioning and SOC synthesis to have the better prototype. The chapter is useful to the prototype and test engineers to have the understanding of the design partitioning across multiple FPGAs and how to achieve the better performance for the SOC prototype.

## Reference

1. [www.synopsys.com](http://www.synopsys.com)

# Chapter 13

## Design Integration and SOC Synthesis

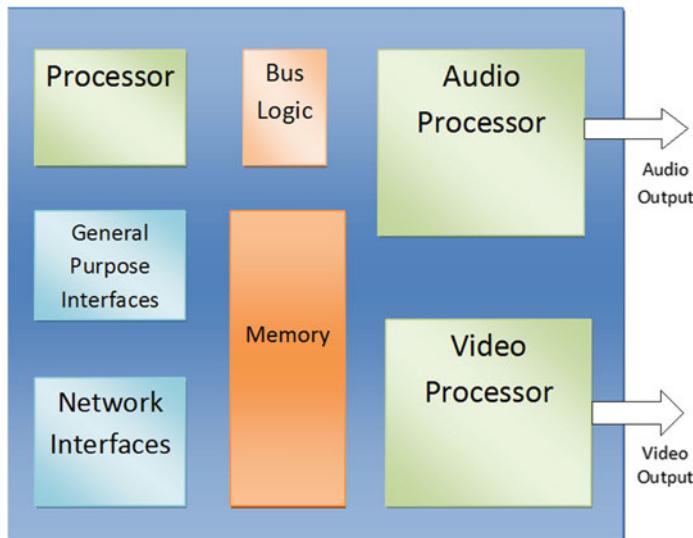


*For the multiple FPGA prototypes, find the number of FPGAs using the initial gate estimation and by using the FPGA architecture.*

**Abstract** This chapter discusses about the SOC synthesis and the design partitioning. To have the better prototype of the SOC as we know that we can have the multiple FPGA architectures. Under such circumstances, the better design partitioning can result into the high performance to have the proof of concept. The chapter key focus is to address the important aspects while partitioning the design. How to overcome the partitioning challenges and how to use the synthesis, place and route, and STA tools with incremental approach to validate the complex SOC designs are also discussed in this chapter!

**Keywords** Partitioning · Hardware and software partitioning  
Multiple FPGA designs · Synthesis · Incremental synthesis · Manual partitioning  
Automatic partitioning · P and R · FPGA resources · IO pads · EDA tools  
Back-end tool

As discussed earlier in previous few of the chapters, the SOC designs in this century are more complex and need the million logic cells to validate the design. Under such scenario, it is not possible to have the FPGA prototype for the SOC using single FPGA. If the multiple FPGAs are used in the prototype, then the prototype and design team need to work together to achieve the best possible performance for the SOC design. The chapter discusses about the SOC architecture, design partitioning, challenges in the design partitioning, synthesis, and how to overcome them!



**Fig. 13.1** SOC block diagram

### 13.1 SOC Architecture

Consider the SOC design used for the multimedia application. The design has the multiple processors. These processors are used to perform the data transfer operations and other algorithm executions. The audio processor is responsible to generate high-quality audio output, and the video processor is used to get the high-resolution video.

Apart from the multiple processors, the SOC has the memory controllers to transfer the data from the external memory and the bus interface logic. The other blocks are network interfaces and general-purpose interfaces and used to interface the external devices with the SOC (Fig. 13.1).

### 13.2 Design Partitioning

For the larger SOCs, the design needs to be partitioned into multiple blocks. This can be achieved at the architecture level, at synthesis or netlist level. What we need to think is the following:

1. Try to have the understanding of the architecture and micro-architecture, and in the iterative way, try to partition the design to achieve the better performance.
2. Have the partition of the design by describing the hardware and software boundaries. For example, the initial setup and configuration can be controlled by the

software; large data storage can be implemented using software. DSP, audio, video, and processing blocks can be implemented using hardware.

3. Try to have the use of the automatic partitioning tools available in the industry.
4. Partition the design at the synthesis or netlist level for better performance.
5. Try to understand the target FPGA resources while partitioning the design to yield the better performance.
6. Have a rough gate count estimation of each functional block in the form of the FPGA resources.
7. Identify the external connectivity and the data transfer speed.
8. Have the larger density blocks in the separate module, and try to partition them to have the better mapping.
9. Have the partitioning by considering the multiple clock domain and power domain designs.
10. Partition the clock and reset network and use the clock and reset network to have the least and uniform skew across the FPGA fabric.
11. Try to identify the external connectivity with the FPGA using high-speed IOs.

### 13.3 Challenges in the Design Partitioning

The design partitioning and the efficient coding play an important role in the SOC design. Most of the time, engineers partition the design depending on the functionality, but this type of partitioning without consideration of the resources required can result into inefficient synthesis.

Proper design partitioning can improve the boundaries between the different functional blocks and enhance the synthesis results. Even the proper partitioning can improve the time during compilation and even can improve the synthesis optimization and overheads on the constraints.

If the logical partitioning is correct, then it not only helps the RTL designer to code the efficient RTL but also it helps to improve the area and top-level timing for the design.

*How we can partition the design for synthesis?*

Large or complex designs can be partitioned into different modular blocks as it can improve the team efficiency during the RTL design and verification phase.

Mainly, the logical partitioning by retaining the same functionality for the design can promote the design reuse during the design cycle. For complex designs, the team members can manage the block-level RTL design and synthesis provided that the design is partitioned in the better way.

This will be useful in the later stage of design cycle to meet the timing requirements for the design.

Following can be few of the techniques which can be used to partition the design to achieve the better performance:

1. Partition the design to promote the design reuse during HDL synthesis.
2. Always keep the required combinational logic in the same block. Do not partition the design at the combinational boundaries.
3. Partition the design at the top level into separate IO pads, core logic, and boundary scan.
4. Always isolate the state machines and state machine controllers from other logic.
5. Do not add the glue logic at the top level
6. Always isolate the synchronizers used for the clock domain crossing.
7. Before partitioning the design, consider the chip layout.

For the efficient synthesis, the designer can use the following guidelines:

#### 1. **Design should be technology independent**

The designer should keep in mind that the HDL should be written in such a way that it should be technology independent. It is possible only when the hardcore instances of library gates are minimized. Try to understand the difference between the instantiation and inference! The preference during the HDL design should be given to inference instead of the instantiation.

The major advantage of the inference of the logic is the design can be implemented for any ASIC library and new technology through the resynthesis. If synthesizable IP cores are used in the design, then the technology independent HDL can improve the synthesis result. Manage the instantiated logic depending on the use of library into separate module, so that it will be lesser time consuming while migrating to other technology library.

#### 2. **How I can partition the clock-related logic in the design?**

Use the separate module for the clock gating logic and reset logic and should be set as *don't touch*. This will help in the clean timing related with the clock module group.

In single module, avoid the use of the multiple clocks as this will help while writing the block-level constraints with reference to the clock.

In the multiple clock domain designs, it is not possible to keep the multiple clock logic in different blocks. In such scenario, perform the stand-alone synthesis of the synchronizers and use the *don't\_touch attribute* while instantiating the synchronizers in the main block.

In the hierarchical designs, use the same name for the clock throughout hierarchy as it helps during script writing and during the synthesis.

#### 3. **Partitioning Challenges in the single and multiple FPGA designs?**

Whether it is single FPGA design or multiple FPGA designs, the design partitioning has many challenges; few of them are listed in this section

1. Design synchronization.
2. Identifying the large density blocks.
3. Identifying the highly interdependent functional blocks.
4. Grouping the interdependent blocks.
5. Identifying the hardware and software functional blocks and their synchronization.

6. Interconnectivity between the multiple FPGAs.
  7. Partitioning of the design with the goal to minimize the interconnect issues.
  8. Signal integrity issues for the multiple FPGAs.
  9. IO voltage domains and connectivity.
  10. FPGA available resources and limitation on pin count.
  11. Clock network delay and uniform distribution of the clock.
4. **What exactly we are trying to achieve with better partitioning?**
1. Better design mapping on multiple FPGAs.
  2. Better IO constraints at FPGA level and at board level. Use of the constraints directly by the placement and routing tool.
  3. Logic replication or duplication for the multiple FPGA designs and efficient use of IO multiplexing.
  4. Better use of the combination of synthesis, partitioning, and placement and routing tools to get the desired performance.
  5. Useful to identify the correct topology in which the multiple FPGAs should be connected.

## 13.4 How to Overcome the Partitioning Challenges

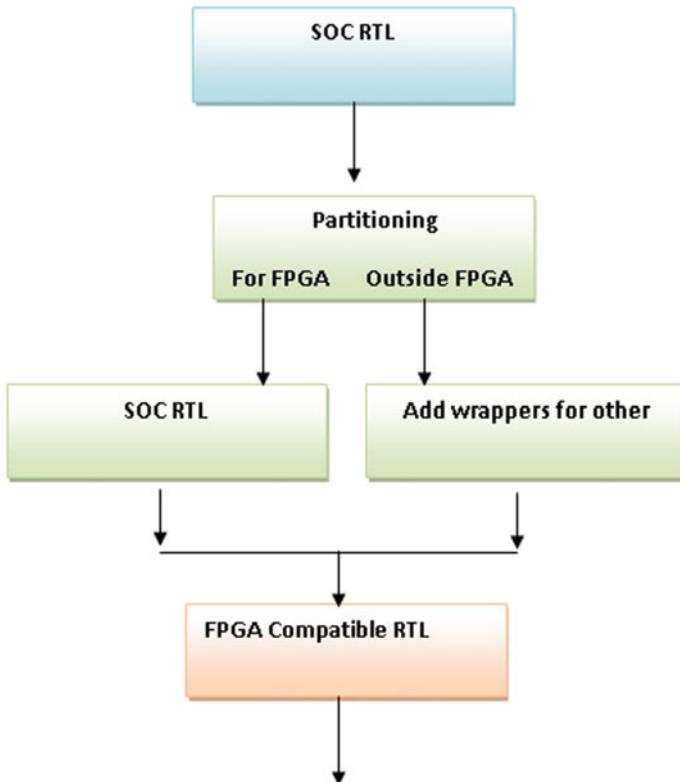
The different techniques at the architecture level and at the netlist level can be used to partition the design for the better prototype. The better partitioning can be achieved by using the partitioning tool. This section discusses about few of partitioning techniques.

### 13.4.1 *Architecture Level*

Have the better understanding of the architecture of the SOC functional specifications and the key resources. The closer look at the architecture and micro-architecture can give the fair understanding of the partitioning of the design at the hardware and software levels.

For example, if we are designing the SOC which consists of the general-purpose processor and DSP processor with audio, video, and other associated logic, then the following way I will think?

1. What are the features of the processor, and what is rough estimation of the resources required?
2. What are the functional specifications of the DSP processor with the audio and video decoders and the resource requirement?
3. Other associated logic such as memory controller, serial-parallel bus logic and internal memory capacity and specifications.



**Fig. 13.2** Design partitioning to get the FPGA compatible RTL

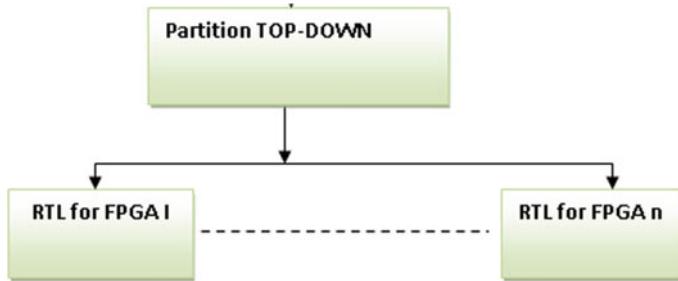
So while partitioning the design, the group I can be processor, memory controller, buses, and internal storage and another group can be DSP processor, audio, and video decoders. This can be a better way while prototyping the SOC using the multiple FPGAs. Again the group I and group II functionality need to be partitioned into the smaller blocks to have modular design.

If we try to speak in the context of prototyping the SOC RTL, then the design can be partitioned by identifying:

1. Design blocks which should be validated using the FPGAs (For the FPGA validation)
2. Design blocks interfaced to the FPGA (outside FPGA).

The design partitioning to get the FPGA RTL is shown in Fig. 13.2.

The FPGA compatible RTL can be partitioned using the top-down approach to get the RTL required for the multiple FPGAs (Fig. 13.3).



**Fig. 13.3** Top-down partitioning for the multiple FPGAs

### 13.4.2 *Synthesis or Netlist Level*

If the design is not partitioned at the architecture level, then the design can be partitioned at the synthesis level. But this is not the better approach for the complex design.

For the moderate gate count design, this may work after performing the synthesis. During the initial synthesis, we can understand about the area and resource requirement to realize the design. To have the better prototyping, using the least area or less FPGA resources is one of the important challenge. Under such circumstances, the design can be partitioned by examining the area reports, block functional dependency, connectivity between the different blocks and clock, and voltage domain analysis.

Better way in this type of approach is to use the area report generated during the synthesis and find out the blockwise report using the partitioning tool. Main important task needs to be accomplished by partitioning tool is to find the IOs for each block and their connectivity to other blocks.

The tool like Synopsys Certify [1] can give information about the resources and IO count. What tool does is that; it overestimates the resource requirements, but gives correct IO count [1]. Most of the time, overestimation of the resource requirement is better to choose the multiple FPGAs as during the later stage it is at least sure that design will be realized (Fig. 13.4).

## 13.5 Need of the EDA Tools for the Design Partitioning

As discussed earlier, the design complexity of the SOC is very high and to have the cost-effective prototype, the option can be used to have plug and play interface boards with the FPGA. This can give the quick and efficient prototype solution, if the timing at the interface level is matched. But the serious trouble in the complex design can be observed, if the design does not fit into the single FPGA. Under such circumstances, it is essential to partition the design into multiple FPGAs.



**Fig. 13.4** Partition at the netlist level

The serious partitioning problems may be due to logic density, speed, multiple clocks in the design and timing/synchronization issues. There are different ways into the design can be partitioned into multiple FPGAs may be at the architecture level or at the netlist level. The better visibility we can get if the design is partitioned at the netlist level because we have the information about the area estimation for the design from area reports.

In the present scenario, if we consider the multi-million gate SOC, then the half or one-third of the design can be fitted on the larger FPGA. The major consideration while designing prototyping platform is the FPGA equivalence of the logic. The FPGA may not have the sufficient BRAMs and DSP blocks to accommodate the logic. Under such circumstances, it is essential to find the number of FPGAs required for prototyping. The pin count of ASIC is always more than FPGA, so the biggest bottleneck is the pin availability and timing. Another important point to be considered

is tweaking of the ASIC RTL into FPGA equivalent. For example, gated clocks, clock enable and ASIC memories need modifications.

The partitioning can be manual or automatic, and the following are few highlights and considerations:

### ***13.5.1 Manual Partitioning***

Most of the time, we try to partition the design manually. Consider the older design and its environment and allow fewer changes to upgrade the design. Under such circumstances, as the older design is proven, the manual partitioning can be cost effective. As there is no additional investments in the partitioning tools, the overall time and budget of such platform is lesser as compared to the automatic partitioning. Following are the important points need to be considered while partitioning design manually:

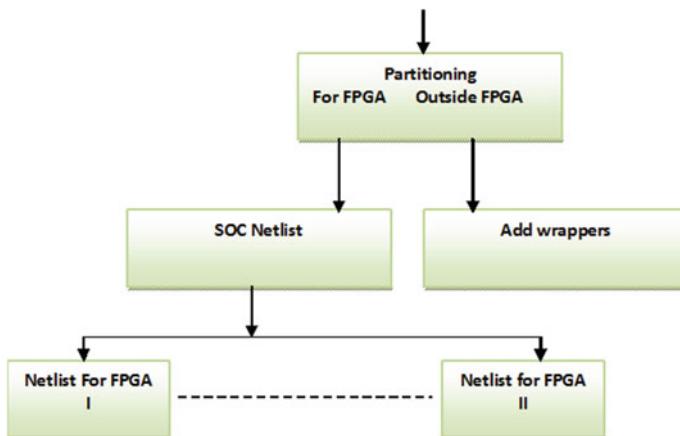
1. Have the understanding of the design architecture and micro-architecture. Keep the timing critical designs close to each other.
2. Have the floorplan to understand the data, control paths, and other interface boundaries for the design.
3. Understand the FPGA resources in much more details and then partition the design into multiple FPGAs. If the resources of FPGA are limited and design is partitioned into multiple FPGAs, the prototype cannot yield into the desired performance.
4. If the enough number of IOs are not available, then use multiplexing and take care that the additional logic should not be placed at the output of MUX.
5. Use the available registers in the IO block to have the sequential boundaries.
6. Use the gated clock conversions and ASIC memory conversions at the RTL level.

The manual partitioning approach is suitable for the moderate gate count designs up to 100 K gate logic. But if the design has millions of the gate and the multiple clock/power domains then manual partitioning is never the cost-effective solution. The manual floorplanning for the multi-million design is not the good solution as it is always error prone. It is not possible to keep the record of the RTL tweaks and conversions for the manual partitioning (Fig. 13.5).

### ***13.5.2 Automatic Partitioning***

Most of the EDA tool vendors like Synopsys offers the automatic partitioning tools and the following should be few of the features used for the automatic partitioning of the design:

1. Partitioning tool should support TCL commands. If it supports Synopsys Design Compiler (SDC) commands then the ASIC scripts/constraints can be used.



**Fig. 13.5** Design partitioning

2. It should be able to define the probe points for the verification of the design. It should allow the integration with the debug tools like Xilinx ChipScope Pro, Intel-FPGA signal Tap and Synopsys Identify [1]!
3. Partitioning tool should support the FPGA prototyping board and the environment.
4. The tool should be able to optimize for the area requirements.
5. The tool should be able to understand the pin requirements and optimize for the number of pins.
6. It should be able to accommodate the small design changes at the netlist level.
7. It should be able to accommodate the large number of design changes at the RTL level.
8. It should give the resource estimation quickly for the target FPGA to ensure that design can fit on the FPGA.
9. It should identify the high-speed IO and clock lines and utilize them to improve the timing
10. Should allow the allocation of the logic to every FPGA. Maximum limit for the better prototyping can be 60–70% of the available FPGA resources.
11. It should give the signal to trace assignment report for the detailed analysis of the prototyping.

Still there are limitations to identify the black box IP and BRAM and DSP inference. So most of the time we observe combined use of the manual and automatic partitioning.

## 13.6 Synthesis for the Better Prototype Outcome

AS SOC designs are faster than the FPGA and logic density is larger the design partitioning for the million gate SOC is the most important task. The design can be partitioned before synthesis or after synthesis. The prototype team needs to choose the correct approach for partitioning the design.

The truth is; design may not run at the SOC speed and it is essential to modify the SOC design into FPGA equivalent resources. So during the synthesis it is essential to have clarity about the architecture or initial floorplan, constraints and FPGA resources. Prototyping flow should achieve the better performance as compared to SOC emulation and for that the major milestone is synthesis. There are multiple ways in which the synthesis can be performed to achieve the better results. The following are few of the approaches used during the synthesis.

### 13.6.1 Fast Synthesis for Initial Resource Estimation

If we chose the fast synthesis then it can be useful for understanding the initial or rough device utilization and the performance at the initial stage. But in such type of synthesis the full optimization is ignored by the synthesis tool. The reason being the runtime is almost around two or three times. But this can be useful to save the weeks/days time for the complex designs and for the initial design partitioning.

### 13.6.2 Incremental Synthesis

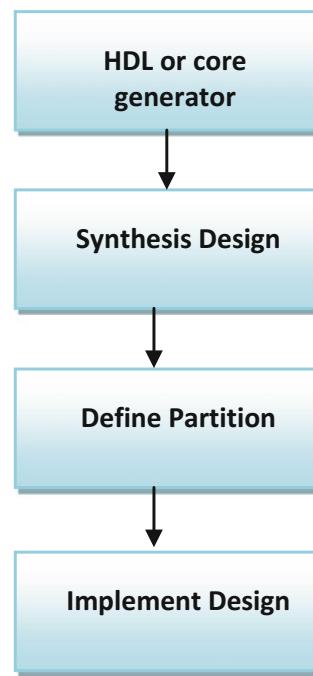
The incremental synthesis is the better approach for the complex SOC designs. The incremental efforts of P and R tool can be used efficiently while synthesizing the larger density designs. The SOC design sub-blocks or trees can be synthesized separately according to the version changes.

For example, consider the SOC design having 100 sub-blocks and the RTL changes are incorporated in only 10 sub-blocks; then during increment synthesis the tool can synthesize the RTL for the 10 sub-blocks. This reduces overall efforts and time during the synthesis phase.

That is if the sub-block or tree architecture is not changed then synthesis tool ignores this and preserves the previous version. This reduces the weeks/days time for the complex SOC synthesis.

The beauty of the EDA tool like Synopsys Certify [1] or the XILINX P and R tool [2] is that they preserve the hierarchy, previous version logic, placement, constraints, and mapping as it is during the resynthesis, if the RTL is not modified. It reduces the turnaround time.

**Fig. 13.6** Design synthesis and implementation



If small portion of the design is modified, then due to incremental synthesis, the design runtime reduces, and P and R tool can use the synthesis results.

The prototype team should be able to use the features of the synthesis and P and R tool. The combined use of these features can reduce the significant amount of time during prototyping. Most important point is that the P and R runtime is always larger than the synthesis runtime for the complex designs. So the strategy should be use the synthesis and P and R tool in the incremental flow (Fig. 13.6).

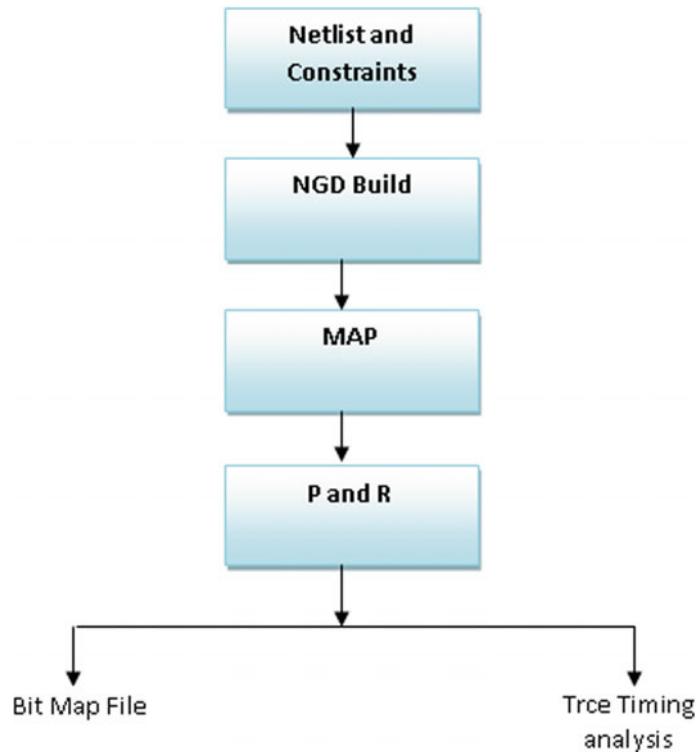
The Xilinx EDA tool back-end flow is shown in Fig. 13.7

## 13.7 Constraints and Synthesis for FPGA Designs

The section discusses about the use of Synopsys DC tool for the FPGA synthesis. The FPGA synthesis commands are listed in Table 13.1.

The following steps can be followed for the FPGA synthesis using Synopsys DC

1. Read the Verilog/VHDL design file
2. Set the design constraints
3. Insert the pads
4. Perform the design synthesis
5. Execute the replace\_fpga command
6. Write the database



**Fig. 13.7** Xilinx Back-end tool flow [2]

**Table 13.1** Commands used during FPGA synthesis

Command	Description
set_port_is_pad <port_list> <design_list>	The command is useful to place attributes on the list of ports specified in command. Attribute allows dc to map IO pads
set_pad_type <type of pad> <port_list>	The command is used to choose the type of the pads to which design is to be mapped
insert_pad	The command is used to insert the pads
replace_fpga	The command is used to convert the synthesizable FPGA database to the schematic, instead of visualizing the schematic having CLBs, IOBs the schematic consists of gate

The sample script for the FPGA synthesis of top\_processor\_core is shown below:

```

dc_shell > read -format verilog top_processor_core.v
dc_shell> create_clock clock -name clk -period 10
  
```

```
dc_shell> set_input_delay 2 -max - < list all the input ports using the same
command and required delay attribute>
dc_shell > set_port_is_pad
dc_shell> insert_pad
dc_shell> compile -map_effort high
dc_shell> report_timing
dc_shell> report_area
dc_shell> report_cell
```

The timing report consists of the timing path information and the data required time, data arrival time, and slack.

Area report gives the list of following:

- Number of ports*
- Number of cells*
- Number of nets*
- Number of references*
- Combinational area*
- Non-combinational area*
- Net Interconnect area*
- Total cell area*
- Total area*

To get the information about the FPGA resources, the following command can be used:

```
dc_shell> report_fpga -one_level
```

It gives the following information about the use of FPGA resources:

- Function Generators:
- Number of CLB
- Number of ports
- Number of clock pads
- Number of IOB
- Number of flip flops
- Number of tri state buffers
- Total number of cells

To write the netlist in the database format, use the command

```
dc_shell> write-format db -hierarchy -output top_procesor_core.db
```

The synthesizable database (netlist) and timing information can be used by the place and route tool.

## 13.8 Important Takeaways and Further Discussion

The following are few important points to conclude this chapter:

1. Use the mix of the manual or automatic partitioning for the complex SOC designs.
2. Use the incremental flows during the synthesis for the quick turnaround.
3. Use the EDA tools to partition the design.
4. Use of the partitioning tool allows the better design mapping on the FPGAs and IO timing.
5. Use the EDA tool directives to get the better performance during the synthesis.
6. Use the IO and speed constraints during the synthesis and propagate them to the back-end tool.

The next chapter discusses about the interconnect delays and timing.

## References

1. [www.synopsys.com](http://www.synopsys.com)
2. [www.xilinx.com](http://www.xilinx.com)

# Chapter 14

## Interconnect Delays and Timing



*To minimize the pin count, use the IO multiplexing.*

**Abstract** This chapter discusses the high-speed interconnects and their need in the design. If we consider about the complex SOC architecture, then the prototype using single FPGA is not always the feasible solution. The prototype needs to have the multiple FPGAs and the connectivity between them can be visualized using the bus topology. The high-speed interconnects between them can reduce the onboard delays and in turn improves the design performance. This chapter focuses on all these aspects, issue, challenges, and solutions to have the high-speed FPGA prototype using multiple FPGAs. The IO multiplexing, time budgeting, and interconnectivity between the FPGA are described using the practical considerations and design scenarios.

**Keywords** Interconnect · Interface · High speed · Onboard delay · Cable Switch matrix · IO bandwidth · Hyper-register · HyperFlex · Pin multiplexing · IO multiplexing · SERDES · LVDS · Deferred interconnect · Star connection · Ring connections

The interconnects between the multiple FPGAs decide the overall speed of the prototype. Most of the times during the board bring-up stage we observe the issues in the performance of the prototype. For complex SOCs, it is the truth that, onboard delays between the multiple FPGAs and other interfaces limit the overall performance of the design. The chapter discusses all these aspects in more details.

## 14.1 Interfaces and Interconnects

The interface between the multiple blocks plays important role in the prototype. The interconnect delays between the processor, IOs; memories need to be reduced as they decide about the performance of the prototype.

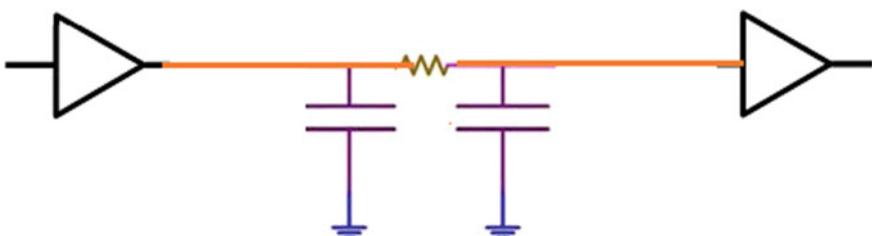
Let us consider Fig. 14.1 in which the output buffer inside the FPGA 1 drives the input buffer using direct interconnect between the FPGAs. The interconnect has the impedance and the wire interconnect model can be visualized as RC circuit. The speed of the data transfer is limited due to the interconnect properties and the RC effect where R is resistance of the wire, and C is the stray capacitance.

Due to the charging and discharging of the stray capacitance, the speed of the data transfer is limited. At higher frequencies, the interconnect between two blocks or devices acts as transmission line. The termination impedance of every interconnect plays important role, and most of the time we need to match the termination impedance. The issues like crosstalk signal integrity degrade the overall design performance at the system level. The care should be taken by the system and board designer to have the least interconnect delays.

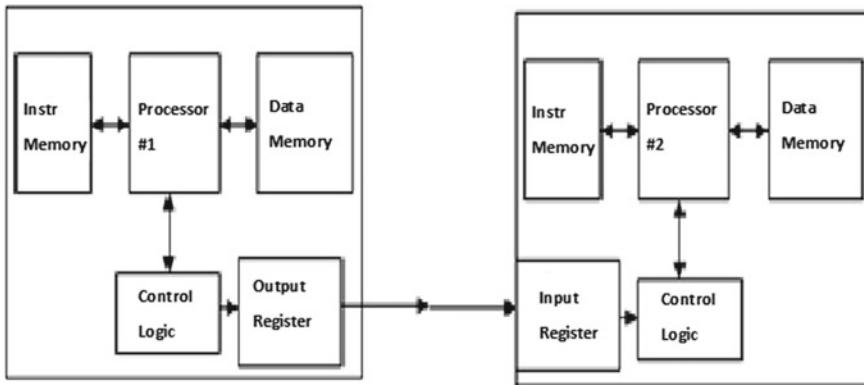
The external chipset interfaces and timing are crucial, and it is another bottleneck in the SOC design.

Consider Fig. 14.2 in which the two identical copies of the processor and associated logic are mapped on the FPGAs. Due to the gate count and resource availability limitation of the single FPGA, the multiple FPGAs are used and the connectivity is established between them. The registers at the IO are used to hold the data. The launch FPGA stores data in the output register, and the capture FPGA uses input register to hold the data.

The direct interface has the limitations in the data transfer due to the FPGA IO pin count and due to the design complexity. As SOC pin count is larger as compared to FPGA pin count, the pin multiplexing need to be incorporated to minimize the pin count.



**Fig. 14.1** RC delays



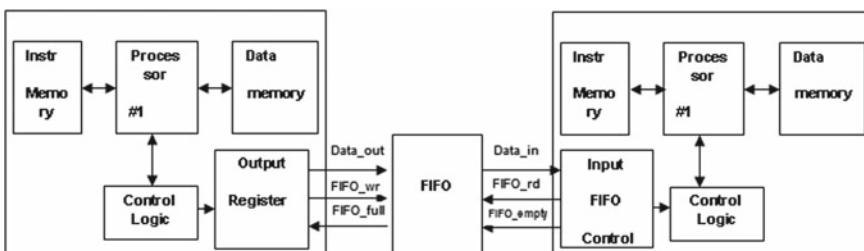
**Fig. 14.2** Identical FPGA blocks and connectivity

## 14.2 Interface for High-Speed Data Transfers

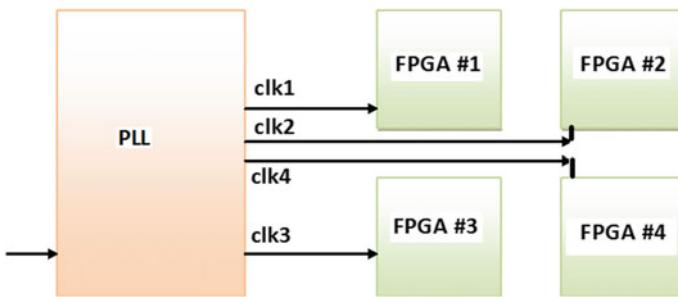
To transfer the data between multiple FPGAs, the better way is to queue the data in the buffer having the required depth. As shown in Fig. 14.3, the FIFO is used to transfer the data between the two FPGAs. The launch FPGA can output the data; write signal is generated when FIFO is not full. The capture FPGA reads the data by generating the read request when FIFO is not empty.

This type of mechanism can add significant amount of delay due to the latency at the read and write side. Even adding the FIFO or circular buffers outside FPGA is not better solution, as it reduces the speed of the data transfer.

For improved performance, the FIFO logic can be implemented inside the FPGA, and using the direct interconnections, the data can be transferred between two FPGAs.



**Fig. 14.3** Interface between two processors using FIFO



**Fig. 14.4** Multiple FPGAs on the board

### 14.3 Interfaces for Multi-FPGA Communication

As discussed earlier, the SOC gate count during this century is almost around 20 million gates and the single FPGA solution is not the better option. Under such circumstance, the prototype should be flexible enough and should have the adequate IO interfaces. As we know that components placed on the single board have less number of challenges during the testing. At the SOC architecture level, the decision should be made about the prototype features requirement. Always it is better choice to consider about the IO speed, IO voltage, bandwidth, clock and reset network, external interfaces, while designing the prototype using the single or multiple FPGAs!

Under such circumstances for the better prototyping, let us understand the connectivity between the multiple FPGAs. Inter-FPGA connectivity is one of the important factor and decides about the performance and quality of the prototype. Consider the Virtex-7 (XC7V2000T) FPGA having the user IO of 1200 and differential IO of 1152 for FLG1925, for the prototype using the multiple FPGAs the connections between the multiple FPGAs or associated peripherals is the key to achieve the desired performance. Multiple FPGA on board is shown in the Fig. 14.4.

Whether to use the ring-type arrangement or star topology to establish connectivity between the multiple FPGAs is the important question need to be addressed! Few prototype engineers may feel that the better option is mixed interconnects using the ring-type connections and star.

#### 14.3.1 Ring-Type Connectivity Between FPGAs

In such type of arrangement, the multiple FPGAs are connected to form the ring.

In such type of connectivity, it increases the overall path delay. As the signal is passing through the FPGA, the equivalent prototype logic can resemble to priority logic. This type of the connectivity has slower speed as compared to other type of boards.

If we try to visualize the ring-type connections, then at high level we can think about the pin connection using such type of inter-FPGA connectivity. The wastage of IOs cannot be limited in such kind of the connectivity. For the FPGAs which are at the downside; IOs will be wasted, and it is additional overhead to the board designer and board layout team to connect these IOs to high-impedance states.

### **14.3.2 Star Connectivity**

This type of inter-FPGA connectivity is faster as compared to the ring arrangement due to the direct connections with the other FPGA. For the better prototype performance, use the high-speed interconnects between the FPGAs and configures the unused pins as high-impedance state.

### **14.3.3 Mixed Connectivity**

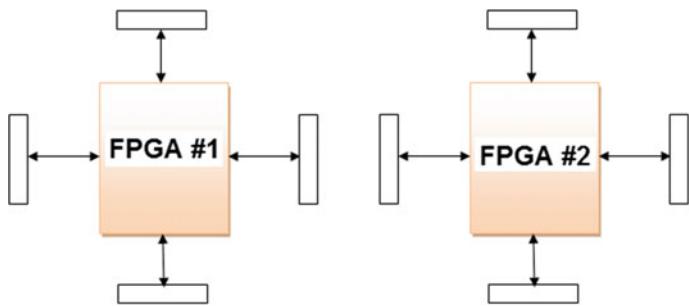
During the board design and layout, we may use the mix of the ring type connections and star connectivity. Such type of connectivity can have the moderate performance. The boards available in the market from vendors have fixed connectivity and may not be suitable during prototyping as they do not match the specifications and requirements. Under such circumstances depending on the design complexity, it is better to choose interface connectivity for better prototype performance.

## **14.4 Deferred Interconnects**

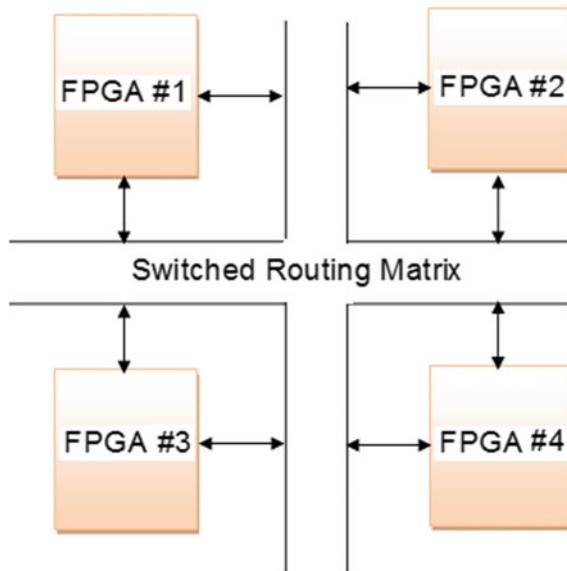
It is very much advisable to use the deferred interconnect using the cables in the multiple FPGA platforms. As connections are not fixed, it gives flexibility to the prototype team to use the cables with the required connections. Figure 14.5 shows multiple FPGA connectivities using the cables.

As shown in Fig. 14.5, the switched routing matrix is used to establish the connectivity between the different FPGA boards. The board layout and design engineer need to provide this type of connectivity for the multi-FPGA design. The number of layers for the complex SOC boards can be of minimum 40–50, and these connections can give the freedom to the prototype team due to programmable features.

This type of connectivity uses the programmable switches with the switch matrix for establishing the connections between the FPGAs. So instead of having the static connections, the board connections can be programmed and configured. Such type of connectivity can be treated as dynamic connectivity with reuse.



**Fig. 14.5** Deferred connection between the FPGA

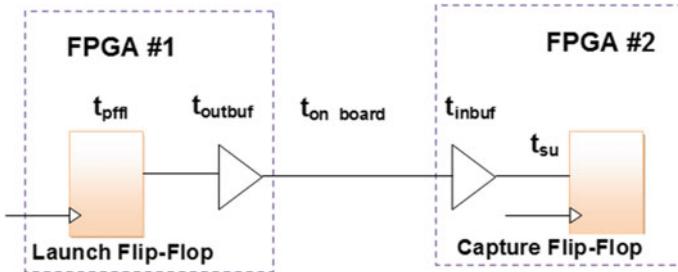


**Fig. 14.6** Multiple FPGAs connectivity using switch matrix

One of the major advantages of this type of connectivity is to add the debug and test circuitry as and when required on the field or off the field due to plug and play programmable arrangement.

The care should be taken to fix the pin placements and during design partitioning to program the switches, and for that purpose, the partitioning EDA tools can be used.

The speed is the important aspect for any kind of prototype, and as discussed the speed of prototype is dependent on the interconnect impedances and stray capacitances. The interconnect delays are function of the wire length and may be prone to crosstalk due to the differed impedances. During the timing estimations, it is required to consider the delays between the FPGAs (onboard delays) and logic delays (on-chip delays). In the simple terminology, we can treat the delays as off-FPGA and on-FPGA delays. To improve the performance of the prototype, it is recommended to use the high-speed differential signals between the FPGAs (Fig. 14.6).



**Fig. 14.7** Multiple FPGAs design timing

## 14.5 Onboard Delay Timing

Let us consider the direct interconnect between two FPGAs. We need to consider about delays to find out the overall data transfer between the two FPGAs.

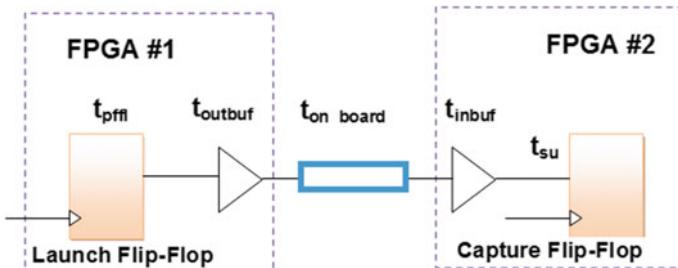
As discussed we can have the direct connection between the FPGAs and following are the main important delay parameters which can be used to find the speed of the design.

- $t_{\text{pff}}$  Clock to  $q$  delay of the launch flip-flop.
- $t_{\text{outbuf}}$  The output buffer delay at launch FPGA pad.
- $t_{\text{inbuf}}$  The input buffer delay at capture FPGA pad.
- $t_{\text{su}}$  Setup time of the flip-flop.
- $t_{\text{on\_board}}$  Onboard delay of the interconnects between the FPGA.

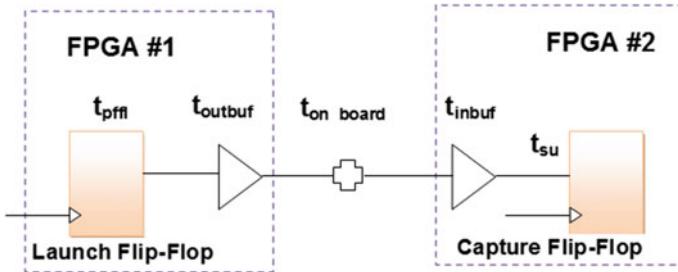
So for the desired timing performance, it is essential to meet the setup time at the capture edge. So the max delay time to reach data at the data input of the FPGA#2 (Fig. 14.7) should be

$$t_{\max} = t_{\text{pff}} + t_{\text{outbuf}} + t_{\text{on\_board}} + t_{\text{inbuf}}$$

Thus, the maximum frequency can be computed by using



**Fig. 14.8** Multiple FPGA communication using cable connectors



**Fig. 14.9** Multiple FPGA communication using switch matrix

$$f_{\max} = (1/t_{\max}) = (1/(t_{\text{pffl}} + t_{\text{outbuf}} + t_{\text{on\_board}} + t_{\text{inbuf}}))$$

If cable connections are used between the multiple FPGAs, then while calculating the maximum operating frequency consider the maximum onboard delay as function of the cable length. The max cable delay between the FPGA can be treated as onboard delay (Fig. 14.8).

$$f_{\max} = (1/t_{\max}) = (1/(t_{\text{pffl}} + t_{\text{outbuf}} + t_{\text{cable}} + t_{\text{inbuf}}))$$

If the programmable switches are used to establish the connectivity between the FPGAs, then use the delay of the switch matrix (Fig. 14.9).

The maximum operating frequency can be computed using the following formula

$$f_{\max} = (1/t_{\max}) = (1/(t_{\text{pffl}} + t_{\text{outbuf}} + t_{\text{switch\_matrix}} + t_{\text{inbuf}}))$$

In such type of connectivity, the onboard delay is denoted by  $t_{\text{switch\_matrix}}$ .

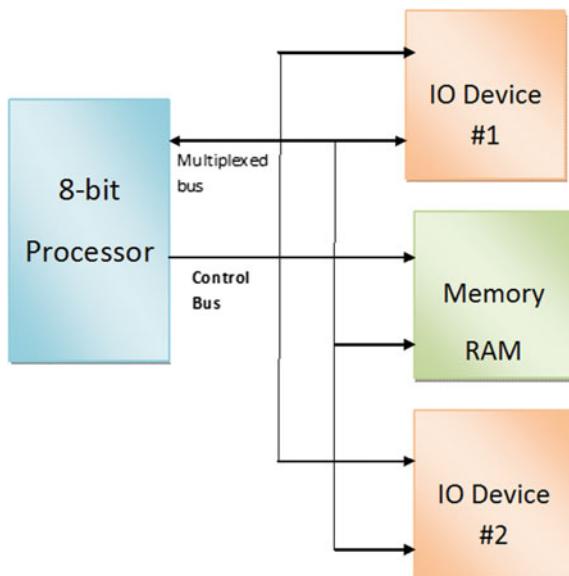
To conclude this; prototype engineer needs to take care of the following important points during SOC prototyping!

1. RTL coding style and the design mapping on the single or multiple FPGAs
2. Complexity of buses and interconnects
3. IP block use and their timing
4. Overall % utilization for each FPGA device
5. IO speed and bandwidth to have the data transfer between the FPGAs.

## 14.6 What Care We Should Take While Designing the Interface Logic?

In the system design, the processors can be interfaced with the external IO and memory devices. As stated earlier to reduce the pin count, the IOs need to be multiplexed. There are few multiplexing techniques such as

**Fig. 14.10** IO multiplexing of address data bus



1. IO multiplexing using MUX and DEMUX
2. IO multiplexing using SERDES
3. Shifter-based IO multiplexing.

The IO multiplexing depending on the design requirements can be incorporated using the RTL or using the EDA tools. The better partitioning and pin mapping tool need to be used for the pin multiplexing.

As shown in Fig. 14.10, the address and data bus are multiplexed. To have the correct address decoding, the bus should be demultiplexed at the destination logic. The IO multiplexing techniques are discussed in Sect. 14.8.

## 14.7 IO Planning and Constraints

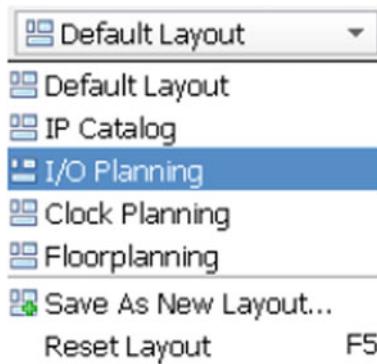
For the efficient prototype, the IO planning, documentation, and constraining them are the important tasks. IO planning using Xilinx Vivado is described below.

Use the IO planning layout shown in Fig. 14.11.

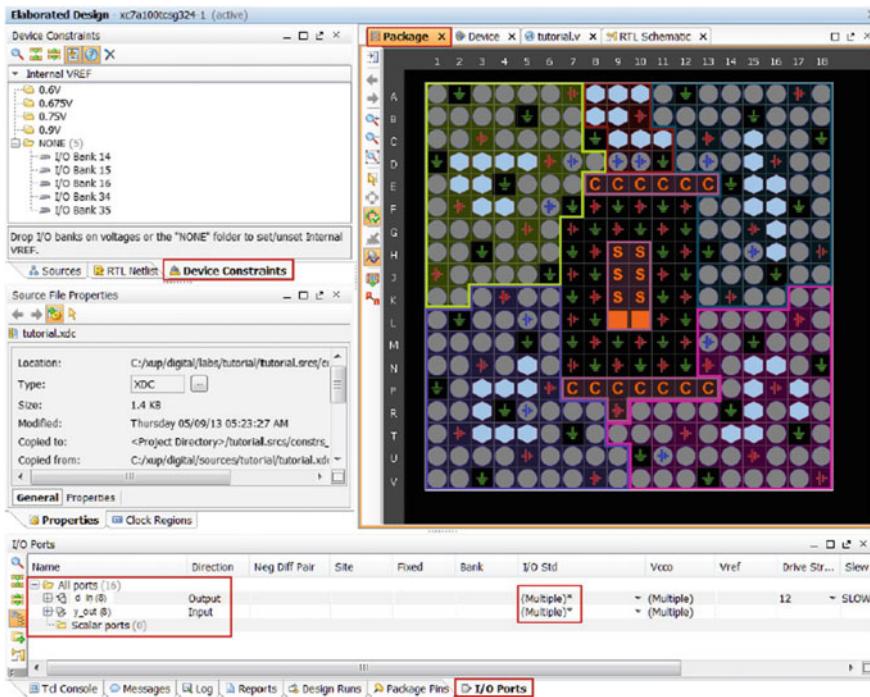
To perform the IO planning, use the following auxiliary view after clicking on IO planning (Fig. 14.12).

In the auxiliary view, the package is displayed, and after selection of the device constraints, the IO ports are displayed in the console area. With multiple IO standards, the design inputs and outputs are listed in the IO tab area.

In the IO tab area, click on the (+) box for inputs (`d_in`) and output (`y_out`) (Fig. 14.13).



**Fig. 14.11** IO planning layout [2]



**Fig. 14.12** IO planning auxiliary view [2]

Now you can see the IO standards. For the d\_in (6 down to 0) and y\_out (6 down to 0), the IO standard LVCMS33 is used, and for the d\_in (7) and y\_out (7), the default IO standard LVCMS18 is used. Depending on the IO requirements, one of the IO standards can be chosen. Now to change the IO standard for the y\_out (7) to LVCMS33, use (Fig. 14.14).

Name	Direction	Neg Diff Pair	Site	Fixed	Bank	I/O Std	Vcco
All ports (16)							
d_in(8)	Output					(Multiple)* default (LVCMS18) LVCMS33*	(Multiple) 1.800 3.300
d_in(7)	Output					LVCMS33*	3.300
d_in(6)	Output					LVCMS33*	3.300
d_in(5)	Output					LVCMS33*	3.300
d_in(4)	Output					LVCMS33*	3.300
d_in(3)	Output					LVCMS33*	3.300
d_in(2)	Output					LVCMS33*	3.300
d_in(1)	Output					LVCMS33*	3.300
d_in(0)	Output					LVCMS33*	3.300
y_out (8)	Input					(Multiple)* default (LVCMS18) LVCMS33*	(Multiple) 1.800 3.300
y_out (7)	Input					LVCMS33*	3.300
y_out (6)	Input					LVCMS33*	3.300
y_out (5)	Input					LVCMS33*	3.300
y_out (4)	Input					LVCMS33*	3.300
y_out (3)	Input					LVCMS33*	3.300
y_out (2)	Input					LVCMS33*	3.300
y_out (1)	Input					LVCMS33*	3.300
y_out(0)	Input					LVCMS33*	3.300
Scalar ports (0)							

Fig. 14.13 IO standards [2]

Name	Direction	Neg Diff Pair	Site	Fixed	Bank	I/O Std
All ports (16)						
y_out(8)	Output					LVCMS18
y_out(7)	Output					LVCMS18
y_out(6)	Output					LVCMS12
y_out(5)	Output					LVCMS15
y_out(4)	Output					LVCMS18
y_out(3)	Output					LVCMS25
y_out(2)	Output					LVCMS33
y_out(1)	Output					LVTTL
y_out(0)	Output					MOBILE_DDR
d_in(8)	Input					PCI33_3
d_in(7)	Input					

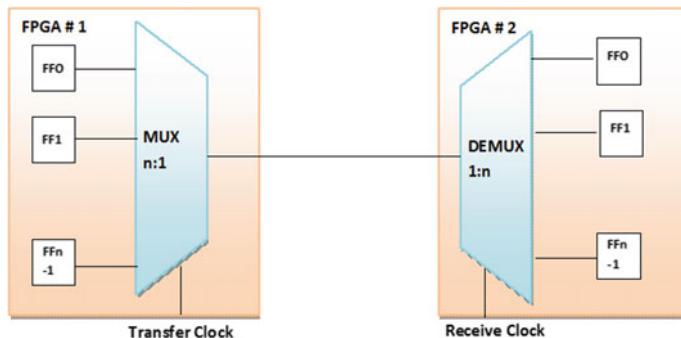
Fig. 14.14 Selection for IO standard [2]

By using the tcl commands, also IO standards can be assigned. Use the following commands

```
set_property package_pin V5 [get_ports {y_out[7]}]
set_property iostandard LVCMS33 [get_ports [list {y_out[7]}]]
```

Even by using the IO port properties, the IO standards can be assigned. After assignment of IO standards, save the constraints in the comb\_design.xdc file.

But for the larger SOC design, the manual IO planning is not the correct options. The manual error can occur while selecting the IO standards, voltage domains and while assigning the constraints. So use the scripting to lock the IO locations for the respective IO standards and to constraint the IO delays.



**Fig. 14.15** MUX-based IO multiplexing

## 14.8 IO Multiplexing

To minimize the pin count, the IO pins can be multiplexed. The section discusses the different IO multiplexing techniques used in the SOC prototype.

As discussed earlier, the IO multiplexing can be achieved by using the

1. MUX-based IO multiplexing
2. SERDES-based IO multiplexing
3. IO multiplexing using the shifter

### 14.8.1 MUX-Based IO Multiplexing

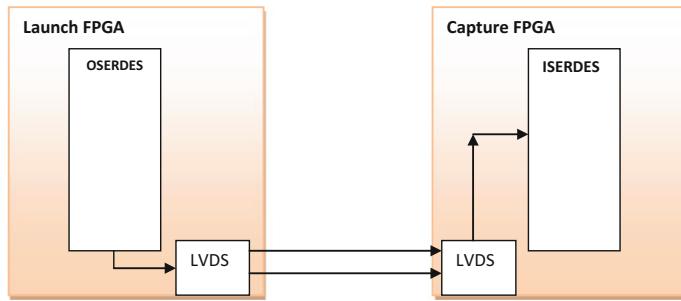
In this type of technique, the multiplexer and demultiplexers are used. Consider the n:1 MUX receiving the IO signals inside FPGA #1. The FPGA #1 operates on ‘Transfer Clock’. The n IOs are multiplexed and transferred by using ‘Transfer Clock’. The FPGA #2 uses ‘Receive Clock’ to receive the sample the IO signals, and they are demultiplexed using 1:n demultiplexer (Fig. 14.15).

If  $n = 4$ , then the 4:1 IO multiplexer need to transfer the IO signal at the clock rate of the  $n * 4$ . So if we consider the FPGA#1 system clock is ‘clk’, then the transfer clock should be  $n * clk$ . The transfer and receive clock in such type of technique should be same.

### 14.8.2 IO Multiplexing Using SERDES

This is one of the techniques to multiplex IOs, and in this technique, the SERDES and LVDS are used to transfer the IO signal from the launch FPGA to capture FPGA.

The launch FPGA can transfer the differential signal using the transfer clock, and this signal is received by capture FPGA (Fig. 14.16).



**Fig. 14.16** LVDS and IO SERDES for serial transfer

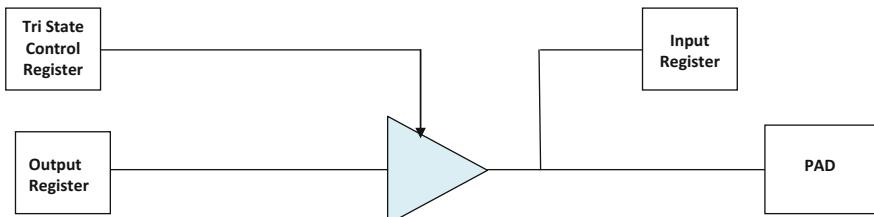
## 14.9 IO Pad Synthesis for FPGA

AS FPGA tools does not understand about the instantiation of the pads; hence, it is essential to modify them during the prototype. As it does not handle the IO pad in the RTL and infers the FPGA pad. So need to leave the pads out with dangling connections inactive or to the top-level boundary. For the prototype, replace each IO pad instance with synthesizable model of FPGA equivalent.

The model should have the logical connections at the RTL level and that can be done by writing small piece of code using the Verilog RTL. For the efficient prototype, prepare the SOC pad library. The basic IO cell for the FPGA is shown in Fig. 14.17.

Use the following commands using Synopsys DC. For more information refer to Chap. 13 for the FPGA synthesis.

```
dc_shell>set_port_is_pad
dc_shell>insert_pad
dc_shell>compile -map_effort high
```



**Fig. 14.17** FPGA basic IO cell

**Table 14.1** Intel FPGA Stratix 10 interconnect [1]

Intel Stratix 10 GX/SX device name	Interconnects		PLLs		Hard IP PCIe hard IP blocks
	Maximum GPIOs	Maximum XCVR	fPLLs	I/O PLLs	
GX 400/SX 400	392	24	8	8	1
GX 650/SX 650	400	48	16	8	2
GX 850/SX 850	736	48	16	15	2
GX 1100/SX 1100	736	48	16	15	2
GX 1650/SX 1650	704	96	32	14	4
GX 2100/SX 2100	704	96	32	14	4
GX 2500/SX 2500	1160	96	32	24	4
GX 2800/SX 2800	1160	96	32	24	4
GX 4500/SX 4500	1640	24	8	34	1
GX 5500/SX 5500	1640	24	8	34	1

## 14.10 Modern FPGAs IOs and Interfaces

The modern FPGA Intel Stratix 10 has the high-speed IOs. Table 14.1 gives information about the general purpose and high-speed interconnects, PLL I/Os and the PCI express hard blocks. Depending on the IO and logic density requirements, the prototype boards can be chosen or designed.

## 14.11 How This Discussion Is Helpful During SOC Prototyping?

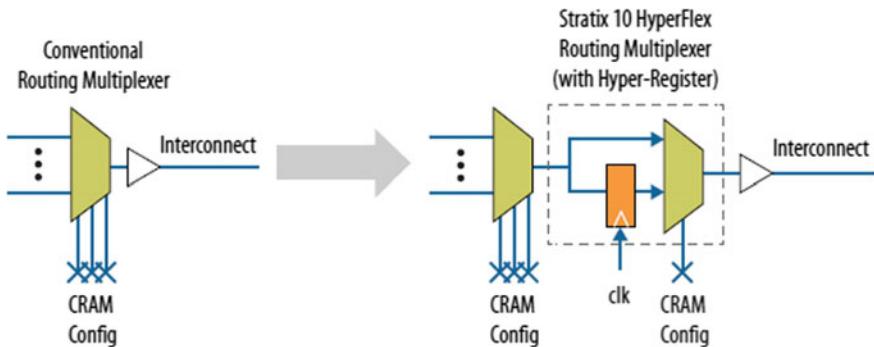
During the SOC prototyping, we need to think about the following points to choose the FPGAs having high-speed IO connectivity and architecture.

1. High-speed IO connection
2. High-speed differential signals
3. Low-voltage differential signals
4. High-speed Interfaces
5. At the implementation level better routing

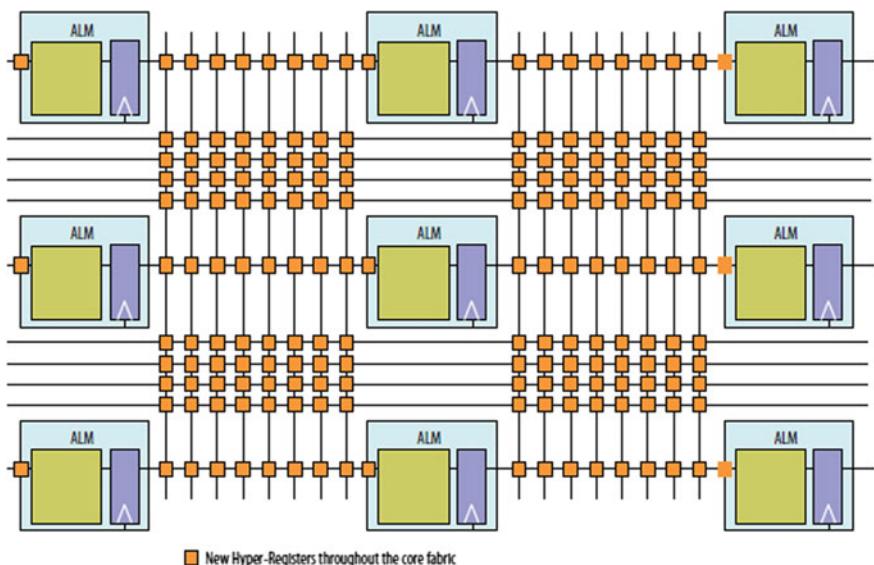
Consider the Stratix 10 HyperFlex routing multiplexer shown in Fig. 14.18.

Intel FPGA Stratix 10 devices have HyperFlex core architecture, and it delivers the 70% lower power compared to the previous generation. The performance is 2X of the clock frequency. The advantage is to have the higher throughput, improved power efficiency, and greater design functionality with improved productivity.

The important reasons are, due to HyperFlex core architecture, the IP size is reduced, and due to 2X clock frequency, the bus width reduces. This feature improves



**Fig. 14.18** Stratix bypass hyper-registers [1]



**Fig. 14.19** Hyper-register throughout FPGA fabric [1]

the routing congestion as a lot of the FPGA resources can be free. Even this improves the overall timing for the design.

Additional by-passable registers throughout the FPGA fabric are available in addition to the traditional registers in the ALM. These hyper-registers are available at the input of the functional block, interconnect, and routing segments.

These registers can be used by hyper-aware tools to improve the design timing performance by reducing the critical path delay and routing delay. Figure 14.19 shows the hyper-registers throughout the FPGA fabric.

## 14.12 Important Takeaways and Further Discussions

As discussed in this chapter, the following are few important points to conclude this chapter

1. The interconnect delays are limiting factor to achieve the performance of the prototype.
2. For the prototype using multiple FPGAs use star, ring or mixed topology.
3. The IO multiplexing is used to reduce the pin count of the design.
4. The IO multiplexing can be implemented by using MUX, shifter, and SERDES.
5. The onboard delays between FPGA need to be minimized to have the better performance of the prototype.
6. The LVDS and SERDES can be used to send IO and clock information from launch FPGA to capture FPGA.

## References

1. [www.altera.com](http://www.altera.com)
2. [www.xilinx.com](http://www.xilinx.com)

# Chapter 15

## SOC Prototyping and Debug Techniques



*Quantum computing is the better choice for the future digital world.*

**Abstract** The chapter discusses the key considerations while choosing the target FPGA and prototyping board to validate the SOC designs. The chapter even covers the multiple FPGA designs and considerations, risk, challenges and how to overcome them. The chapter also covers the Xilinx Zynq-7000 device features and the SOC platform considerations.

**Keywords** FPGA · LUT · CLB · MUX · Device utilization · ROM · RAM · Block RAM · Multiplier · Xilinx FPGA · Intel FPGA · SOC · DSP · Zynq 7000 · Z-7020 · PS · PL · CMT · DDR2/DDR3 · PLL · Ethernet · SPI · I2C · UART · CAN

As discussed in the previous few chapters the million gate SOC designs can be prototyped using the FPGA. The real challenge to achieve the desired performance in such kind of the design is due to the multiple FPGA architecture, design partitioning and the connectivity between them. The chapter discusses about the risk, challenges and how to choose the target FPGA to have the better SOC prototype.

### 15.1 SOC Design and Considerations

In the past few years while working in the field of the FPGA designs, I have observed most of the time that the complex design does not fit on the single FPGA. The important aspect with the designer we thought that let us tweak the architecture to find out is there any room for the resources to fit inside the single FPGA?

It was logical thought as well as the thought to avoid the use of multiple FPGAs. The outcome of such kind of discussion was positive, and with the architecture tweaking and using the synthesis tool directives to use the FPGA resources efficiently we were successful to map the design in single FPGA.

While prototyping for the complex SOCs what should be our approach? In the practical environments, every organization has their standard flows while prototyping the SOCs. The team manager and leader can think about when to start about the prototyping? At which stage? After achieving the desired coverage goals at the block level? Or wait to complete the full functional verification.

All these questions can be answered if we look into the prototype flow! We cannot have the real estimation of the FPGA resources at the start and even it is difficult to differentiate between the logic which can be mapped inside the FPGA and outside the FPGA. As a team, we need to focus on the functional correctness of the design at the start. So team leader should think about the start of prototyping phase after passing all the basic tests at the sanity level. At least, the team members will be sure about the data passing from one functional block to another functional block, and even they will be sure about the basic functional correctness of the design.

If we try to use the waterfall model that is first phase RTL design and second phase RTL verification using the system Verilog and then the final phase as prototype and testing, then this may delay the project by few weeks or months. Instead of using the waterfall model it is better approach to concurrent kick-off RTL design, verification and board bring up phase. The concurrent task execution can achieve the desired milestones in the better way.

### **What are the important tweaks during the prototype using FPGA?**

The important point to consider is that what about the IO pad rings? What about the large external memories? What about the analog blocks? Effectively, we will not be able to map the large external memories, analog blocks, and IO pad rings inside the FPGA!

At the architecture level, it should be decided, which functionality should be tested using the FPGA, and what kind of prototyping board is required to test or emulate the overall SOC.

So, let me put this in the real scenario of prototyping the SOCs using the analog blocks and hard IP cores. Practically, it is not possible to have the FPGA netlist for analog blocks as well as hard IP blocks. In the absolute reality, the IP design houses provide the evaluation boards and the prototype team needs to create the SOC prototype platform with single or multiple FPGAs interfaced with such type of the evaluation board to achieve the desired performance and functionality. Let us discuss the important aspects of such type of designs.

1. **Interface signals:** The FPGA and the evaluation board should have compatible interfaces. If IOs are not compatible, then there will be an issue of the signal integrity.
2. **Connectivity between FPGA and evaluation board:** RTL should provide the connectivity between the design functionality realized on the FPGA board and external evaluation board.

3. **Timing:** It is one of the important aspects due to the interface delays between the FPGA and the external IP board. So need to have clock resource analysis with the inter delay timing.
4. **Synchronization:** For booting, the external hardware may take less time as compared to FPGA logic, so during the prototype care should be taken that the FPGA bitstream should be loaded first inside the FPGA. Again, the synchronous and asynchronous reset/reset removal pulse durations need to be analyzed.
5. **Power supply:** Should have the digital and analog ground isolation and multiple voltage levels depending on the requirements. Even care should be taken that the FPGA and external board connectivity should maintain the desired voltage and logic levels.
6. **Mechanical assembly:** The care should be taken to have the good mechanical assembly so that during the different geographical locations/during transit the board should not damage.

## 15.2 Choosing the Target FPGA

As discussed in Chap. 11, the key FPGA resources are CLBs (to map the combinational, sequential logic), memories, interconnect resources, clocking resources, DSP blocks, IO blocks, and transceivers. Now depending on the need, the FPGA can be chosen for the prototype. What we need to think about the available resources as mentioned above.

**Combinational and sequential logic:** How many CLBs the FPGA has? This can answer the logic density of the FPGA. As the logic (combinational and sequential) is packed inside the CLBs (LUTs + FFs + additional cascade/carry logic), consider the CLB count. For example, considering Virtex7 XC7V200T device it has 19,51,560 logic cells and the logic can be mapped using these logic cells depending on the SOC architecture.

**Memories:** As discussed, focus on the number of BRAMs available and their configuration while selecting FPGA. Each FPGA has multiple BRAMs and configured as RAM, ROM, FIFO. If we consider the Virtex7 XC7V200T architecture, then the device has 18 Kb capacity 2584 memory blocks and if configured as 32 Kb the maximum number of blocks are 1292. The maximum capacity of block RAM is 46512 Kb.

**IOs:** How many IO pins the FPGA has is one of the important points to consider while choosing the FPGA! As discussed, the IO has various standards, drive strengths. For example, the Virtex7 XC7V200T has 1200 user IOs and can be used as differential IO pairs.

**Interconnect resources:** These resources are controlled and used by the back-end tools during the place and route. It is important for the designer to know about what are the different interconnect resources and networks available in the FPGA.

**DSP blocks:** To perform the MAC, shifting, comparison of the signals, filtering, we need to have information about the number of DSP blocks available in the FPGA.

For example, the Virtex7 XC7V200T devices have the 2160 DSP blocks. Due to use of the dedicated DSP blocks, the other resources can be free and can be used to infer some other functionality.

**Special purpose blocks:** While choosing the FPGA if we come to know about the available hard macros such as Ethernet, PCI express, processor cores, SERDES, then it is added advantage.

**Clock resources:** How many dedicated clock generators the FPGA has is another important point need to be considered while selecting FPGA. The programmable clock generator uses the PLLs, clock buffers for global and local connections, and clock skew distribution. The Virtex XC7V200T device has 24 clock manager tiles to provide the clocks with lower skew.

## 15.3 SOC Prototyping Platform

### What I need to think about while choosing the SOC prototype platform?

In the SOC design context as the SOC architectures are complex and may need million gate logic what we need to think about the FPGA resources and whether they will be suitable for SOC prototype or no? Most of the time, we encounter the scenario where the logic cannot fit in the single FPGA and we need to choose the multiple FPGAs! The Xilinx Zynq boards can be suitable to prototype the SOC.

Let us think about a few key points for the SOC prototyping using FPGA!

1. As FPGA is flip-flop-rich logic, we need to think about the ratio of the flip-flops to combinational logic. If the ratio is high, then the design can meet the performance and the timing will be clean.
2. Whether my design is pipelined or not? This can give answer of the maximum achievable clock frequency.
3. The overall resource estimations should be thought in the form of the flip-flops, logic cells, memories for better results.
4. Always need to consider the FPGA gate count estimation which is the approximation and not yield into the exact gate count estimation for the desired SOC functionality.
5. Always, it is better to have the understanding of the clocking resources as that will be useful for the multiple clock domain designs.
6. It is always essential to know about the routing resources as they can give more information on the congestions in the design during the place and route stage. If one of the designs uses the 80% of the resources and other design uses 60% of the resources, then the later design has the less number of routing resources and FPGA will be able to breathe due to use of less power.
7. Last but not least, the important point is the IO pin count for the single FPGA implementation or implementation of the design using the multiple FPGAs. This

is important to map the FPGA logic in case of well-partitioned design, and this can give information about the IO multiplexing and interconnectivity challenges during the early stage of SOC prototyping.

## 15.4 How to Reduce the Risk in the Prototype?

Reducing risk is the important objective, and for that purpose we need to have close observation about the synthesis results and overall resource utilization for the SOC design. At the architecture level, we can have the correct estimation for the use of the external interfaces, memories, DSP blocks, IOs, multipliers.

But it is very difficult or impossible to estimate the number of flip-flops required for the design and even it is difficult to predict the total number of logic cells for the design. In this context, the better approach is to perform the synthesis at the end of the RTL design phase and go through the resource utilization summary. The thumb rule is the resource utilization for the SOC design should be in the range of 60–70% for the single FPGA design. If the utilization is more than 70%, it is better choice to partition the design into multiple FPGAs. That call should be taken by the prototype team.

For the routing debug logic and for the functional specification changes additional logic are required. So while prototyping or selecting the FPGA, do not consume the 100% FPGA resources.

If the resource utilization results into the utilization more than 100%, then definitely the SOC design cannot be mapped on the single FPGA and may need the multiple FPGAs. Although there are architecture and RTL tweaks to reduce the area, they may not yield into the efficient prototype due to high logic cell requirement, routing, and high efforts during synthesis. Instead of that find the available LUTs in the FPGA and take the ratio of the, LUTs required for the SOC Design to the LUTs available in the FPGA.

This ratio is useful to conclude the number of FPGAs. But instead of thinking about the use of 100% FPGA resources, it is better practice to use max of 60% FPGA resources. If I am the team lead, then I will consider the FPGA resource utilization around 40–50%.

So, the number of FPGAs =  $(\text{LUTs required for the SOC Design}) / (\text{LUTs available in the FPGA} * 40\%)$ . This can give the better freedom to the architecture team to cope up for the architecture/design changes and to add the test and debug logic for the SOC design.

For example, considering the Virtex7 7VX200T FPGA has LUTs = 12,21,600 and design needs, for example, 21,00,000 LUTs, then the number of FPGA required is  $\text{FPGA required} = (21,00,000) / (12,21,600 * 0.4) = 4.29$ , so we can think about the use of five FPGAs by incorporating the design portioning.

## 15.5 Prototyping Challenges and How to Overcome Them?

### How I can find the performance of the prototype?

Let us imagine the complex SOC design, and for the efficient prototype we need to have the estimation of FPGA resources. How I can find the correct estimation is one of the challenges. Is there any efficient way to get the accurate FPGA estimations and requirements? Answer at high level is big ‘no’ as stated earlier the estimation of the required resources may not be enough to choose the FPGA. The reason is we need to think about the desired FPGA performance. Let us discuss the key parameters which need to be thought to meet the performance requirements.

Recall that estimating the design resources using synthesis tool can give us information about the ballpark figure. This technique can give information of the performance for the design at high level. The performance capture using the synthesis tool is without the routing delays, but using the placement and routing tool the timing performance can be evaluated. The parameter on which the FPGA performance is dependent is the constraints. The constraints are used by the synthesis, place, and route tool to meet the desired performance.

If I have multiple FPGA designs, then following key parameters affects the performance of the design

1. **Pipelined architecture:** The design without pipelining runs with less speed as compared to the design using pipelined architecture. So, care should be taken to have the multiple pipelined stage controllers for the design.
2. **FPGA device utilization:** If the FPGA utilization crosses almost around 60%, then the design performance slows down. The reason is the congestion in the placement and routing. Even such kind of design has high interconnect delays and hence slower speed. Refer Chap. 14 for the information about the interconnect delays and timing.
3. **Fan-out and load:** The design having high fan-out runs slower as compared to the design having low fan-out.
4. **Synthesis tool and environment:** The use of the synthesis tool to optimize the design is one of the factors which are responsible for the speed of the design.
5. **Inter-FPGA connectivity:** The main important factor to limit the overall speed of the prototype in the multiple FPGA systems. This is due to the inter-FPGA connectivity and IO speed. As most of us know that IO speed is much lower as compared to the speed of the logic on the FPGA fabric.
6. **Multiplexing features:** Pin multiplexing is one of the major factors to limit the speed of the prototype the reason. Consider the practical example of use of the multiplexed IO, and if the  $n$  bit multiplexed logic runs at the operating frequency of 25 MHz, then to sample the  $n$  bit signals the multiplexed input should operate at  $n$  times of 25 MHz.
7. **Delays:** The propagation delays on the board, data rate of individual signals are other factors responsible for limiting the FPGA performance.

## 15.6 Multiple FPGA Architecture and Limiting Factors

As stated earlier, if the design does not fit in the single FPGA, then we need to use the multiple FPGAs to prototype the SOC. Let us think that is there any limit for the use of multiple FPGAs? Theoretically, I can document the design at the architecture level by using multiple FPGAs, but the efficient prototype is that the SOC can be validated using the minimum number of FPGAs and following are the reasons.

**Interconnectivity between multiple FPGAs:** In the multiple FPGA architectures, the interconnectivity depends upon the use of the number of FPGAs. As the number of FPGAs grows in the system, there are issues like signal integrity and large interconnect delays. The system may become slow and may not yield into the desired performance. Under such circumstances, it is recommended to use the fewer FPGAs by tweaking the architecture. To overcome the inter-FPGA connectivity issue, use the time division multiplexing for the IOs using the higher clock frequency. In such kind of techniques, the risk is the clock rate as the multiplexed IOs need to operate at higher clock rate as compared to the logic on the FPGA fabric.

**Design partitioning:** Manual design partitioning is one of the most important bottlenecks in the multi-FPGA system as the designer needs to think about the netlists for the multiple FPGAs and their connectivity. The design partitioning using the tools is the complex task and manual design partitioning is not the feasible solutions in the multi-FPGA system design. In such scenarios the prototype team can think about the mix of the partitioning techniques.

**Propagation of signals and connectivity:** In the multi-FPGA system, another important issue is the settling time required for the signal due to the inter-FPGA connectivity. The IO delays add up the cumulative effect, and it slows down the system.

**Clock generation and clock distribution:** For the synchronous multiple FPGA systems, the clock skew is one of the limiting factors in achieving the desired performance. It becomes additional overhead for the designer to manage the clock distribution on the board to balance the clocking skew.

**Use of multiple licenses during prototype:** Most of the time, we observe that for the multi-FPGA design; the netlist for each FPGA should be generated concurrently. This reduces the overall time required during the prototyping, and it improves the productivity of the team. But this increases the testing, debug, and prototype cost as more number of design engineers need to be employed to achieve the same.

## 15.7 Zynq Prototyping Board Features

As discussed earlier, for any complex SOC prototyping the FPGA should have the features like high speed, parallel processing environment with the resources like DSP, video and audio processing, enough memory for the internal storage. All these features are available in Zynq 7000.

The ARM processor having the required performance and pipelining features with required IO features makes it one of the powerful FPGAs for the prototyping. The key features of Zynq 7000 are discussed in this section.

### 15.7.1 Zynq 7000 Block Diagram

The Xilinx Zynq-7000 block diagram is shown in Fig. 15.1 and as shown it has Extensible Processing Platform (EPP) that is Programmable SOC (AP-SOC). The key features are

1. It has the FPGA fabric with the ARM processor on the single silicon die.
2. FPGA fabric has the programmable logic (PL) and processing system (PS). The PL is based on the Xilinx 7 series architecture.
  - a. If we recall the Xilinx 7 series technology, then we can conclude that PL is based on the Artix-7 or Kintex 7 series fabric which is 28 nm TSMC HPL process.
  - b. It has multistandard IO, gigabit transceivers, and analog to digital converters.

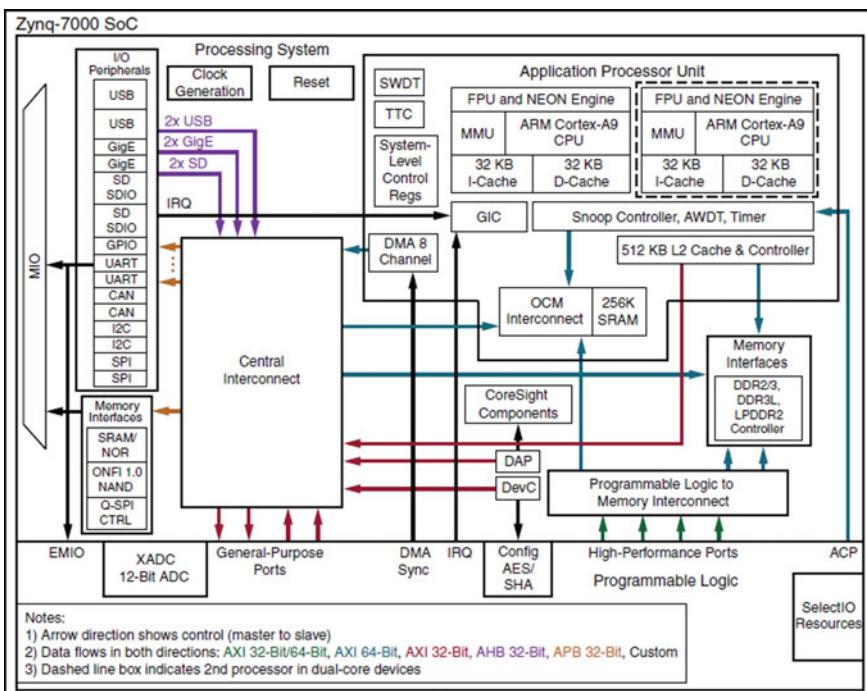


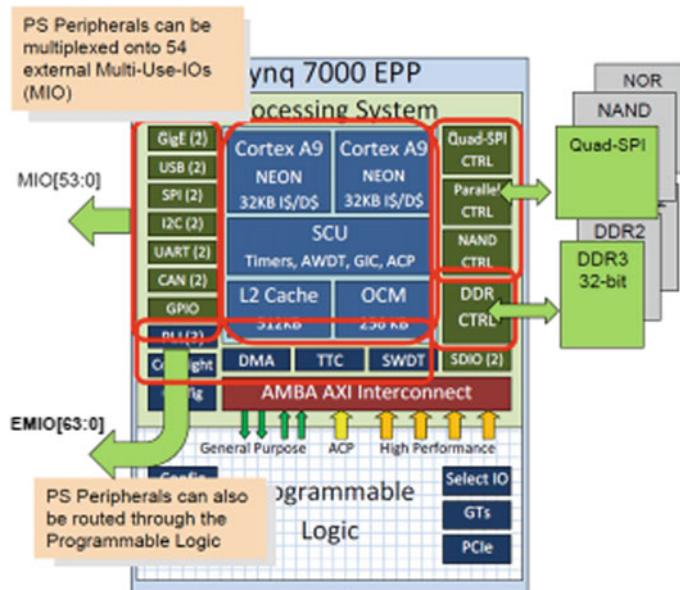
Fig. 15.1 Xilinx Zynq-7000 block diagram [1]

3. PS is based on the dual core Cortex-A9, and it has
  - a. Dual core Cortex-A9 MP core which can operate at 1 GHz frequency.
  - b. It has extended DRAM interface, L1/L2 caches, on-chip SRAMs, and other peripheral interfaces.
4. The SOC prototyping platform is also supported by Xilinx using industry standard tools that is Xilinx/HLS and ARM/Linux.

### 15.7.2 Zynq 7000 Processing System (PS)

Figure 15.2 gives information about the Xilinx Zynq-7000 PS. As shown in the figure, it has the Application Processing Unit (APU) and key features are listed below

1. It has dual core Cortex-A9 Neon with the 512 KB L2 cache.
2. It has Snoop Control Unit (SCU) with L1 cache coherency.
3. It has On-Chip Memory (OCM) that is dual-port 256 KB SRAM.
4. It has external memory interfaces those are DDR2/DDR3 and ECC memory controller.
5. Even the Zynq 7000 PS has the Quad SPI and NAND/NOR flash which can be used during the design configuration.



**Fig. 15.2** Xilinx Zynq-7000 processing system [1]

6. The Zynq 7000 has the peripheral support using the standard IO for PS/PL (2x Ethernet, 2xUSB, 2xUART, GPIO, 2xI2C, 2xCAN, and 2xSPI). Even it has the PLL for the clock, Debug Access Port (DAP), DMA controller, interrupt controllers, and timers.

### 15.7.3 Zynq 7000 Programmable Logic (PL)

The Xilinx Zynq-7000 has PL-PS interfaces, and the key interfaces include Accelerator Coherence Port (ACP) which is for the coherence access to caches. The General Purpose (GP) AXI port has 2x masters and 2x slaves with the connect to central crossbar.

It has high-performance (HP) AXI ports that is 4x master, FIFO buffered, and direct memory access (DMA). It has system interfaces, and the key interfaces include the 16 shared interrupts to GIC, 4 private interrupts to core and debug interfaces. Figure 15.3 gives information about the PL.

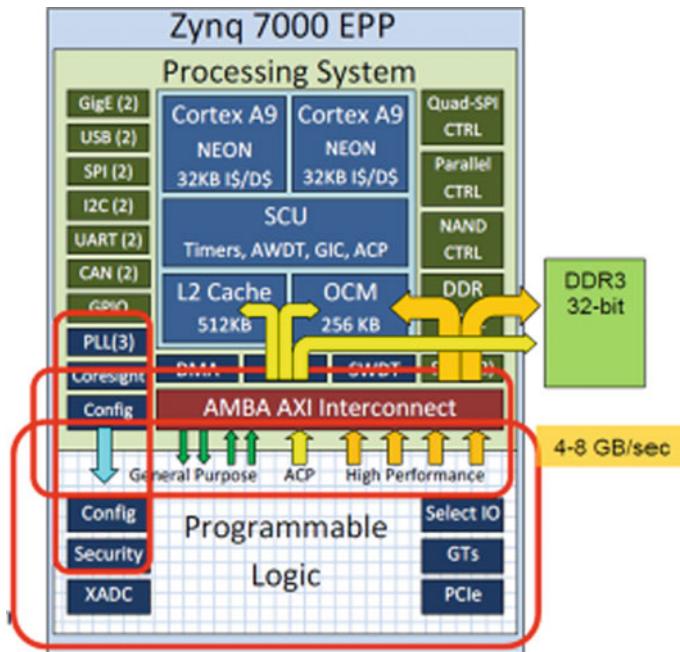
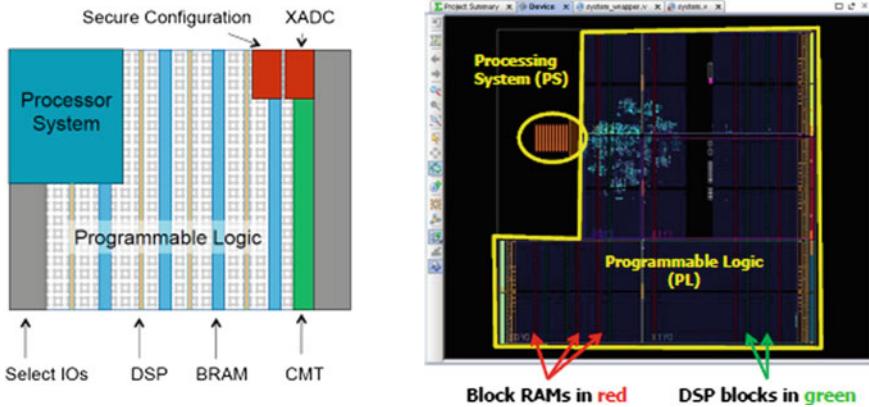


Fig. 15.3 Xilinx Zynq-7000 programmable logic [1]



**Fig. 15.4** Zynq 7000 logic fabric [1]

#### 15.7.4 Zynq 7000 Logic Fabric

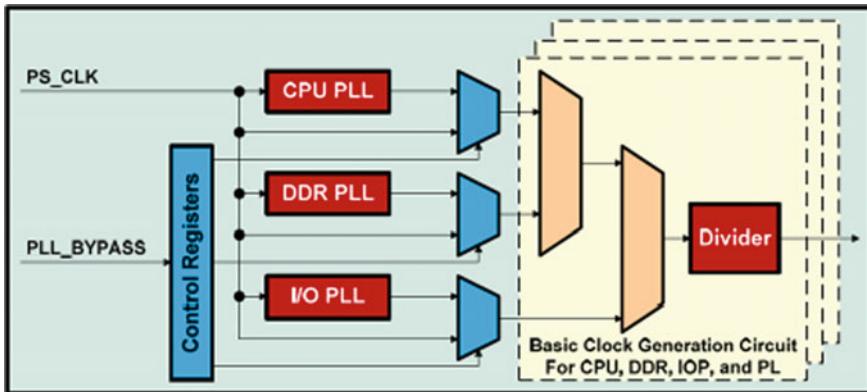
Xilinx Zynq-7000 has the logic fabric which is same of the Xilinx 7 series technology. Figure 15.4 gives information about the logic fabric, and it has the embedded BRAM. DSP slices, CMTs and IOs, PCI Express and A/D interfaces.

#### 15.7.5 Zynq 7000 Clocks

Xilinx Zynq-7000 clock generation module is shown in Fig. 15.5 and it has the PLLs which are used to generate the clock for the CPU, DDR, IO, and PL, whereas PS\_CLK is external 30–60 MHz reference clock. Even the clock generation logic has four general purpose clocks used for PL, and they are named as FCLK\_CLK0, FCLK\_CLK1, FCLK\_CLK2, FCLK\_CLK3.

#### 15.7.6 Zynq 7000 Memory Map

The Xilinx Zynq-7000 memory map is shown in Table 15.1 and it has the 4 GB of addressable memory.



Source: Avnet Inc., "Introduction to the Xilinx Zynq-7000 EPP," Avnet Xtest, 2012

**Fig. 15.5** Zynq 7000 clock generation [1]

### 15.7.7 Zynq 7000 Device Family

The Xilinx Zynq-7000 family has the different capacity devices and is shown in Table 15.2. Low-end devices are 7010, 7015, and 7020, and mid-range devices are 7030, 7035, 7045, and 7100. These devices have different packing options, and even they are available in different speed grades.

**Table 15.1** Xilinx Zynq-7000 memory map [1]

Start address	Size (MB)	Description
0x0000_0000	1,024	DDR DRAM and on-chip memory (OCM)
0x4000_0000	1,024	PL AX I slave port #0
0x8000_0000	1,024	PL AX I slave port #1
0xE000_0000	256	IOP devices
0xF000_0000	128	Reserved
0xF800_0000	32	Programmable registers access via AMBA APB bus
0xFA00_0000	32	Reserved
0xFC00_0000	64 MB–256 KB	Quad-SPI linear address base address (except top 256 KB which is in OCM), 64 MB reserved, only 32 MB is currently supported
0xFFFF_COOQ	256 KB	OCM when mapped to high address space

**Table 15.2** Xilinx Zynq-7000 devices [1]

	Device Name	Z-7007S	Z-7012S	Z-7014S	Z-7010	Z-7015	Z-7020	Z-7030	Z-7035	Z-7045	Z-7100
	Part Number	XC7Z007S	XC7Z012S	XC7Z014S	XC7Z010	XC7Z015	XC7Z020	XC7Z030	XC7Z035	XC7Z045	XC7Z100
Programmable Logic	Xilinx 7 Series Programmable Logic Equivalent	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Kintex-7 FPGA				
	Programmable Logic Cells	23K	55K	65K	28K	74K	85K	125K	275K	350K	444K
	Look-Up Tables (LUTs)	14,400	34,400	40,600	17,600	46,200	53,200	78,600	171,900	218,600	277,400
	Flip-Flops	28,800	68,800	81,200	35,200	92,400	106,400	157,200	343,800	437,200	554,800
	Block RAM (# 36 Kb Blocks)	1.8 Mb (72)	2.5 Mb (50)	3.8 Mb (107)	2.1 Mb (60)	3.3 Mb (95)	4.9 Mb (140)	9.3 Mb (265)	17.6 Mb (500)	19.2 Mb (545)	26.5 Mb (755)
	DSP Slices (18x25 MACCs)	66	120	170	80	160	220	400	900	900	2,020
	Peak DSP Performance (Symmetric FIR)	73 GMACs	131 GMACs	197 GMACs	100 GMACs	200 GMACs	276 GMACs	502 GMACs	1,324 GMACs	1,324 GMACs	2,622 GMACs
	PCI Express (Root Complex or Endpoint) <sup>[3]</sup>		Gen2 x4			Gen2 x4		Gen2 x4	Gen2 x8	Gen2 x8	Gen2 x8
	Analog Mixed Signal (AMS) / XADC	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs									
	Security <sup>[2]</sup>	AES and SHA 256b for Boot Code and Programmable Logic Configuration, Decryption, and Authentication									

### 15.7.8 Zed Board

Xilinx Zynq-7000 prototyping board is shown in Fig. 15.6 and key features are listed below

1. The Xilinx device family is Z-7020, and speed grade is -1. Maximum operating frequency is 667 MHz that is ARM clock, and bus clock is 150 MHz.
2. It has onboard DRAM of size 512 MB, and the memory type is DDR3.
3. It has interfaces for the LED, switch GPIO, Ethernet, USB, UART, and SD cards.
4. It has PL peripherals for the audio, video, and display. Other peripheral modules and FPGA mezzanine card (FMC).
5. It has system control features for the reset, clock, and debugging.

## 15.8 Important Takeaways and Further Discussions

As discussed in the chapter, the SOC can be prototyped using the single or multiple FPGAs and following are key important points to be considered while prototyping the SOCs using FPGA.

1. Use the concepts of the synchronizations of the clock network to distribute the clock across multiple FPGA.
2. Choose the target FPGA to have the desired speed of the prototype.
3. Partition the design in the better way to have FPGA resource utilization of 60–70% for each FPGA.
4. Use the high-end FPGAs like Zynq 7000 to realize the better prototype.
5. Use the high-speed IOs to transfer the data between the multiple FPGAs.

The next chapter discusses the SOC system-level verification and test debug logic.

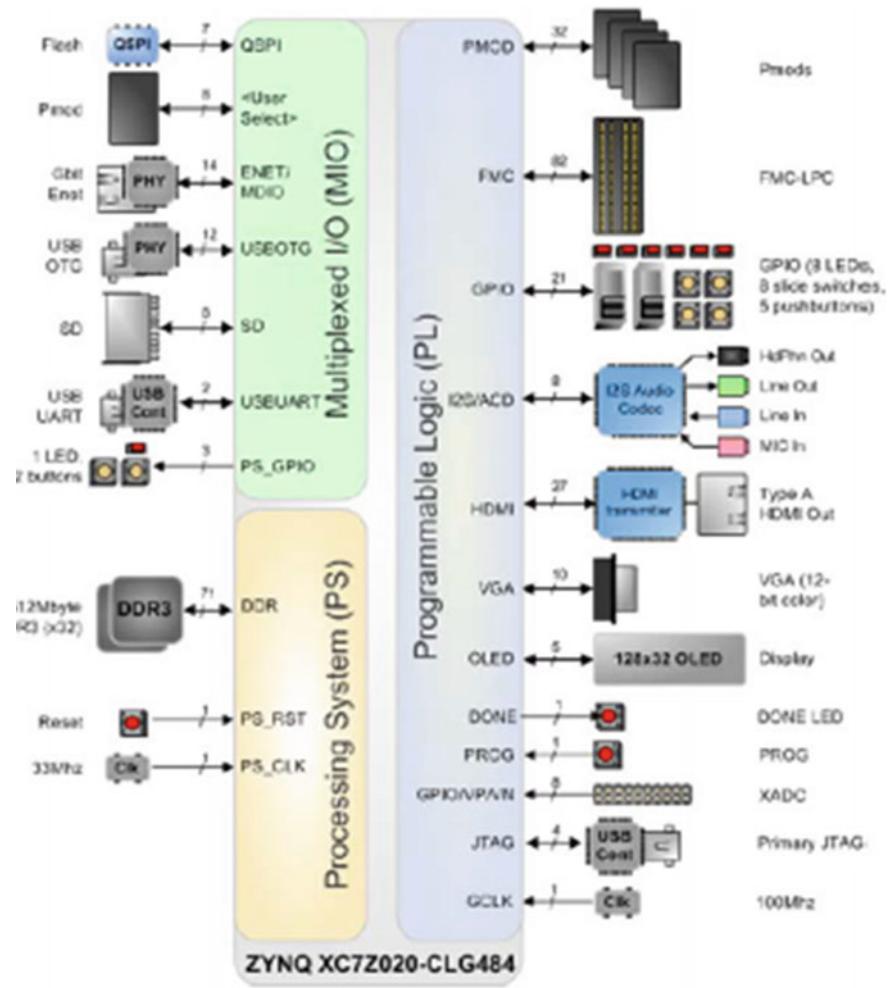


Fig. 15.6 Xilinx Zynq-7000 board features [1]

## Reference

1. [www.xilinx.com](http://www.xilinx.com)

# Chapter 16

## Testing at the Board Level



*For the SOC debugging, use the ILA cores and logic analyzers.*

**Abstract** The chapter discusses the important points useful during the board bring-up stage to validate the SOC design. The chapter covers the debug planning, challenges, board testing for the single FPGA and multiple FPGAs. This chapter can give the understanding of use of the logic analyzer while testing the SOC design. The inter-FPGA connectivity issue, pin and location constraint issues are also discussed in this chapter.

**Keywords** FPGA · IO · Configuration · Multiple voltage domains · IO pins · Pin muxing · LVDS · Timing · Gated clock · Skew · Glitches · Impedance · Signal integrity · Latency · Throughput · Logic analyzer · ChipScope Pro · ILA API · Oscilloscope

The SOC design is ready and the verification is carried out. Now the last important phase is to validate the design on the board. The design can be single FPGA or multiple-FPGA design this need to be validated on the FPGA board. The chapter discusses the board testing, challenges and how to overcome them to have the SOC prototype.

### 16.1 Board Bring-Up and What to Test?

What should be the strategy while testing the million gate SOC? This is the first important question needs to be answered! As a prototype team there should be debug and testing plan in place. The better debugging plan to test the single FPGA design or multiple FPGA design can create the significant amount of the improvement in

the productivity for the SOC design. Following are few important steps need to be followed for the larger SOC designs.

1. Test the board for basic read/write.
2. Test the add-on boards and connectivity.
3. Configure the single FPGA with the small design.
4. Configure multiple FPGAs using the design partitioning.
5. Understand the implementation issues.
6. Use the actual SOC design and test.
7. Check for the issues and fix them.
8. Document the results.

## 16.2 Debug Plan and Checklist

The board bring-up and debug plan need to be efficiently documented during the design planning and the architecture phase. It is not possible that the design downloaded on the FPGA can work right for the first time. It is one of the time consuming tasks to debug the design at the board level. For the single and multiple FPGA designs, the following can be included in the debug plan:

1. Basic test needs to be carried out for the single FPGA designs.
2. Test for the add-on boards interfaced with FPGA.
3. High-speed IO tests.
4. Interface test.
5. Testing and debug using logic analyzer.
6. Multiple FPGA connectivity and debug.

The main objective of all the above is to catch the bugs at the board level. Most of the bugs may not be found during the functional verification, and the better debug plan using the EDA tools, logic analyzers, and transactors can give the more visibility to such kind of the bugs. These bugs at system level can be identified and fixed at the synthesis, *P* and *R*, or at the board level. The following can be few of the guidelines used for debugging the FPGA prototype (Table 16.1).

### 16.2.1 Basic Tests for the FPGA

Run the basic tests to understand the programmability of the FPGA.

1. **Read/write test:** Read and write the FPGA registers, and confirm the FPGA is configured by the correct bitmap file.
2. **Counter test:** Use the existing switches on the FPGA board to check the connectivity, and check for the clock and reset for the programmed FPGA.

**Table 16.1** Few guidelines for the debug

Guideline	Description
Use the smaller design	Use the smaller design to configure the single FPGA at the first time, and carry out the basic read/write tests
Check for the add-on board connectivity	Check for the add-on board connectivity with the FPGA
Check for the IO	Check for the working of the multiplexed IOs
Check for the multiple FPGA connectivity	Check for the multiple FPGA connectivity by partitioning the smaller design across multiple FPGAs
Check for the IO pad configuration	Check whether the IOs are configured in the correct manner or not?
Check for the external world connectivity	Check for the connectivity with the external boards like flash controller interface, DDR interface
Check for the external chipset and IP interfaces	Whether the IPs and the chipset are having the required behavior need to be checked
Use Xilinx IO delay elements	Use programmable IO delays to control the IO timings and to establish the connectivity with the external element
Tweak the clock rate	For the higher clock rate, if the system is not working for the read/write; then to ensure and conclude the issue, reduce the clock rate. If at low clock rate the system responds, then try to debug the issue for higher clock rate
Check for the bus connectivity	Check and confirm the connectivity between the FPGA and logic analyzer for the little and big endian-ness
Check for the termination impedances	Most of the time at the board level design does not work due to wrong termination
High-speed IO test	Check for the high-speed transceivers for the gigabit data transfer, and then test the protocol

### 16.2.2 Add-On Board Tests

Write a small routine to configure the add-on boards interfaced with the FPGA, and confirm that the FPGA can read or write the information from the add-on boards. Debug team can use small routine to configure the multiple registers on the main FPGA board and on the add-on board, and confirm the connectivity and configuration.

### 16.2.3 Test the External Logic Analyzer and FPGA Connectivity

Test the logic analyzer bus working with the main FPGA board. The transfer of the small packet from the FPGA bus can confirm the connectivity.

### ***16.2.4 Multiple FPGA Connectivity and IO Test***

Check for the pin multiplexing and high-speed time division multiplexing. Create the test environment to check for the TDM and data rate at the multiplexed IO pins.

### ***16.2.5 Test for the Multiple FPGA Partitioning***

If the prototype has the multiple FPGAs in the design, then check for the inter-FPGA communication by writing the small design. Small design to ensure the connectivity between FPGA can be partitioned across the multiple FPGAs quickly. This can be small controllers for similar kind of read/write inside the multiple FPGAs.

## **16.3 What Are Different Issues on the FPGA Boards**

Document the major issues on the board and test for them. Few of the issues are listed in Table 16.2.

## **16.4 Testing for the Multiple FPGA Interface**

The following can be the strategies for the design prototyping for the multiple FPGAs:

1. Timing checks for the design partitioned at the system level.
2. The effect of the redundant logic or logic removal on the connectivity across multiple FPGAs.
3. Is the inter-FPGA connectivity intact. This can be tested by read/write transactions across the multiple FPGAs.
4. Whether the gated clocks are mapped into the FPGA equivalent or the design exhibits some issues due to the non-convertible clocking using MUX.
5. Are there any timing violations due to the internally generated clocks?
6. Is the multiplexed interconnects are working at the transfer clock speed or need to tweak the transfer clock.
7. The compatibility of the interconnects and interfaces with reference to the source and sink currents at board level.
8. Whether all the pin locations and constraints are validated.

Consider the practical issue at the board level while communicating between the multiple FPGAs (Fig. 16.1).

Consider the launch and capture FPGA has System\_clk and the design runs at the system clock frequency. The IO signal is launched using the Transfer\_clk and captured at the input of de-mux using the Transfer\_clk. The maximum frequency can be calculated by using

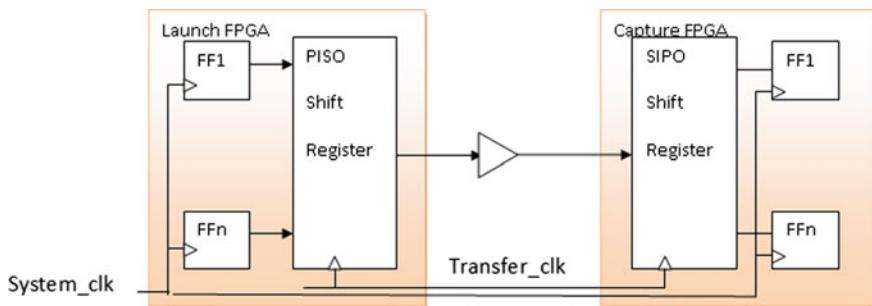
**Table 16.2** The board testing issues and solutions

Debug issue	Type	Description	Action
Design does not work at higher clock	Timing	The design might have the timing violations at high speed, and to confirm that, reduce the clock rate and then check for the design. If the design works at the lower clock rate, then debug	Go through the implementation timing report, and check for warnings. Check for the critical paths and missing constraints
The IO multiplexing does not work	IO speed	This may be due to the improper ratio of the system clock at which design works and transfer clock at which data is transferred	Check for the timing constraints for the transfer clock and system clock. Check for the timing report
Design does not work at low clock frequency	Timing	This may be due to hold violation	Check for the timing reports and constraints, and then fix them by tweaking the min-max analysis
Design does not have the timing violation during $P$ and $R$ , but the design has issue at board level	Board delays	One of the reasons may be the onboard delays are not considered during the timing analysis	Check for the onboard delays from vendor and include them during the timing analysis. Use the safety delay margin or tweak IO delays
The design does not work for the multiple FPGAs having different clock domains	Timing	The issue is metastability and synchronization	Check for the use of synchronization for the multiple clock domain designs. One more reason may be design partitioning at the multiple clock domain boundaries
The design has gated clock and not working on the board	Timing violation	The design may have the hold time violations	Check for the gated clock conversions. Check for the non-convertible clocks and the warning reports
Data are available at the output of launch FPGA pad but not captured in the input register of capture flip-flop	IO configuration issue	This may be issue on board due to IO standards	Check the single-ended IO configuration and differential IO configuration. Make sure the correct IO standards are used during IO mapping

(continued)

**Table 16.2** (continued)

Debug issue	Type	Description	Action
The design works for some conditions but not for all the conditions	IO connectivity	May be due to the improper termination of the IO	Check for the IO impedances at transmitting and receiving end. Check more details in the FPGA vendor-specific digital-controlled impedances
The design works individually on FPGA but not working when configured on the multiple FPGAs	IO connectivity	This may be due to the IO connectivity, IO voltage levels, signal levels	Check for the IO constraints and IO pin mapping. Even check for the IO termination
The FPGA boards have multiple cables, and the design does not work	Connectivity or cable faults	This may be due to termination impedance, delays, timing violation, or signal integrity	Check for the termination impedances for each cable, and then use the vendor-dependent data and fix the issue

**Fig. 16.1** Multiple FPGA IO transfer issue

$$F_{max} = (1/(T_{mux\_delay} + T_{on\_board} + T_{demux\_in}))$$

Where  $T_{mux\_delay}$  = Output delay of multiplexer

$T_{on\_board}$  = On board delay

$T_{demux\_in}$  = Input delay at the capture FPGA

For example consider  $T_{mux\_delay} = 4$  nsec

$T_{on\_board} = 2$  ns

$T_{demux\_in} = 2$  ns

Then  $F_{max} = 1/8$  ns = 125 MHz

For the safer side, take tolerance in the design around 1 ns; then, the maximum frequency for the design is 111.11 MHz.

The question is that whether the design works at this frequency. If you refer Chap. 14, I have stated that IO multiplexing using  $n$ : 1 MUX needs the transfer clock of  $n \times \text{system\_clock}$ . Now, if the system\_clock is 25 MHz then the transfer\_clk should be minimum 100 MHz or maximum 125 MHz. But practically it is not possible to achieve this clock rate to transfer the IO signal.

Practically, the maximum design frequency of such type of multiplexed IO is limited by the clock latency. If the transfer clock frequency is 111.11 MHz, then the System\_clk frequency should be 12.35–15.70 MHz.

So the prototype team should find out the:

1. Maximum transfer frequency using the IO delays, onboard delays, and additional tolerance margin delay.
2. Find the ratio of the system\_clk and transfer\_clk.
3. Give the constraints in synthesis and  $P$  and  $R$  for the transfer\_clk and system\_clk.
4. Give the clock to clock constraints in the synthesis and  $P$  and  $R$  for the transfer\_clk and System\_clk.

## 16.5 Debug Logic and Use of Logic Analyzers

The following section discusses the use of the logic analyzers and few practical considerations while debugging the design.

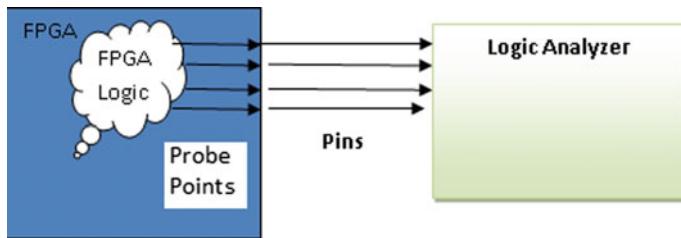
### FPGA Debug:

1. Use feature of the EDA tool to view the internal nodes.
2. Tool should observe the FPGA boundary and the FPGA surrounding logic.
3. Debug and testing, characterization of the high-speed IO: For the fast IOs at 1 MHz, the PC board traces act as transmission line and due to that issue of the signal integrity.

### 16.5.1 Probing Using IO Pins

Use the FPGA IO pins and probe using logic analyzer (Fig. 16.2).

The required signals to probe can be routed at the FPGA pins and can be tested using the external logic analyzer. But for the design which needs the larger number of signals to probe this is not the best solution as there is always limitation on the FPGA pin count.



**Fig. 16.2** Probing using the probe points

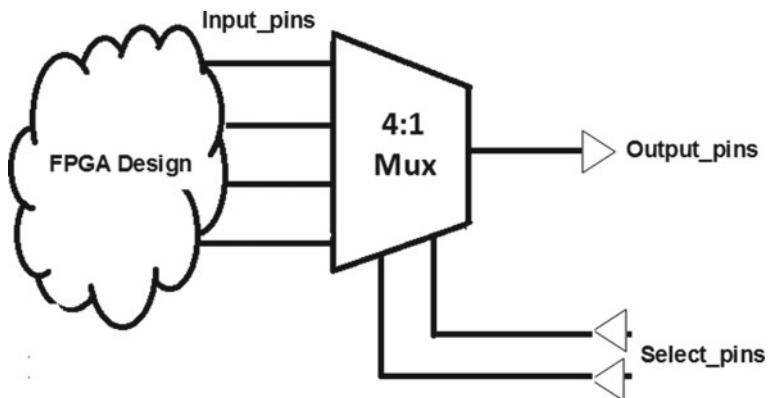
### 16.5.2 Use of the Test MUX

The advantage is that it needs the less number of pins, but the issue is that the probing of MUX output lines is only possible (Fig. 16.3).

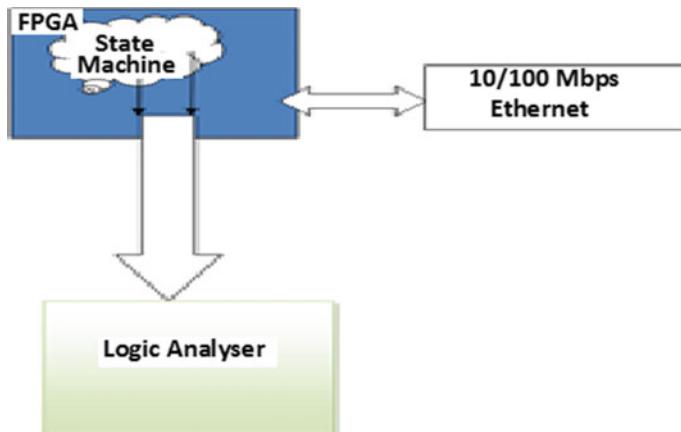
If 16 signals need to probe, then use the 4X 4:1 MUX and two select lines. At the Output\_pins, four signals can be available depending on the status of the select\_pins and they can be used by the external logic analyzer for debug.

### 16.5.3 Use of Logic Analyzer: Practical Scenario (To Detect the Data Packet Is Corrupted)

Consider the FPGA design which has multiple-state machine controllers and the other associated interfaces like Video/Audio CODEC, 10/100 MB Ethernet interface, and other physical interfaces. If the data packet transferred from the Ethernet is corrupted and not identified at the functional simulation, then the overall design has



**Fig. 16.3** Test MUX



**Fig. 16.4** Logic analyzer to detect the corrupted data packet

the bug. Under such circumstances, it is essential to probe the state machine controller designed for accepting the data packets from the Ethernet.

Use the logic analyzer, and try to probe the state machine controller by identifying the state of the state machine controller where packet is corrupted. So look inside the FPGA before the corrupted packet and probe the design. Identify the state where packet is corrupted, and trigger the logic analyzer for probing.

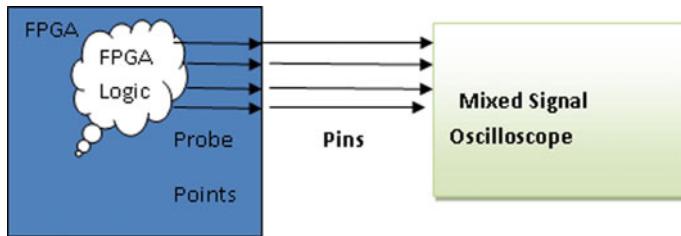
Create the debugging scenario in the simulation, and fix the issue (Fig. 16.4).

#### 16.5.4 Oscilloscope to Debug the Design

Another debug solution is to use the oscilloscope to monitor the signals at the FPGA boundary. It is better to have the oscilloscope to monitor the mixed signals. The analog and digital channels can monitor the behavior of the FPGA and the signal activity at the FPGA boundary.

Consider the DDR controller interfaced with the FPGA, and the address is updated from 01FFH to 0200H but the issue is in the reading of the data. On the previous read address select the data were read into the FPGA from the memory but now the issue is in the read. This needs to be debugged, and under such scenarios, the mixed signal oscilloscope can be used to monitor the behavior of the signal at the FPGA boundary.

By triggering the oscilloscope for this event, the nature of the read address select can be captured. The reason may be the skew, slow transition of the signal, or weak driving strength. This can be fixed at the design level and at the board level (Fig. 16.5).

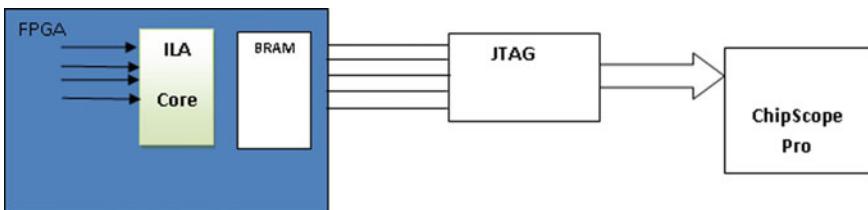


**Fig. 16.5** Use of the mixed signal oscilloscope for the probing

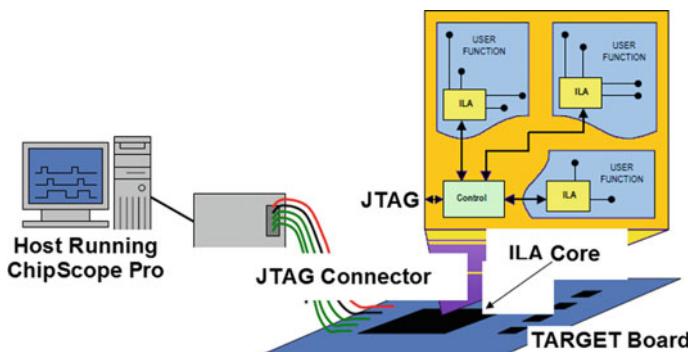
### 16.5.5 Debugging Using ILA Cores

To select many nodes, the better technique is use of the in-built or Integrated Logic Analyzer (ILA) with block RAM (BRAM). This is an inexpensive solution, and no additional pins are required. But it increases the size of the memory inside the FPGA. In this mechanism, the required nodes can be selected and then traces are captured and send to ChipScope Pro via JTAG (Fig. 16.6).

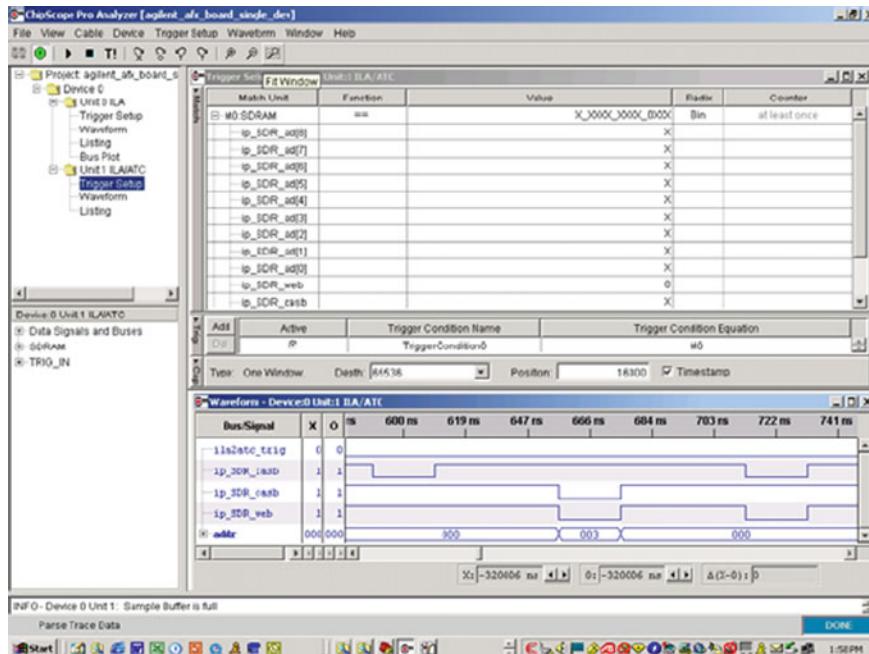
The use of the ChipScope Pro to establish communication via JTAG with the ILA is shown in Fig. 16.7.



**Fig. 16.6** Use of ILA cores for debugging



**Fig. 16.7** Use of Xilinx ChipScope Pro for debugging [1]



**Fig. 16.8** ChipScope Pro view during debugging [1]

There are ways to insert the ILA into the design. They can be instantiated at the RTL level and then synthesize the design, place-route the design, and then generate the bitstream file.

The other way is to insert the ILA blocks depending on the requirements into the presynthesized design and then perform the place-route and generate the bit file. The ChipScope can download the bitstream file into FPGA including the ILA cores.

As shown, the JTAG controller block is interfaced with the ILA cores and the data can be taken out from the ILA cores through JTAG and can be viewed by ChipScope Pro at host PC (Fig. 16.8).

## 16.6 System-Level Verification and Debugging

For the better verification outcome, use the integration of the existing EDA tool environment, software models, and the FPGA prototyping boards. This can be treated as hybrid verification. This allows the overall verification and testing of the FPGA prototype with high verification coverage goals and even the detection of the bugs which were not determined in the early verification phase.

1. There are industry standard cycle-based and event-based simulator available to enable the verification task to have more functional coverage. Even the transactors and transaction-level modelling can be used to verify million gate count design.

Is it possible that if we connect the simulator with the system and it works in the first attempt? Answer is big ‘no’ as there are different scenarios, interfaces, and data transfer issues for the complex SOC design.

The verification and debugging need significant amount of efforts, and it is a time-consuming task.

For the moderate gate count design, the simulation to understand the functional mismatches can serve the purpose. But for the million gate SOC designs the verification, debugging is the rigorous and time-consuming task. Almost around 60–70% of the design cycle time is invested in the verification of the SOC. If we consider for the FPGA prototype, then for the RTL coding just around 10–15% design cycle time is required. And for the remaining phases from the synthesis to the board bring-up it consumes around 85–90% of design cycle time.

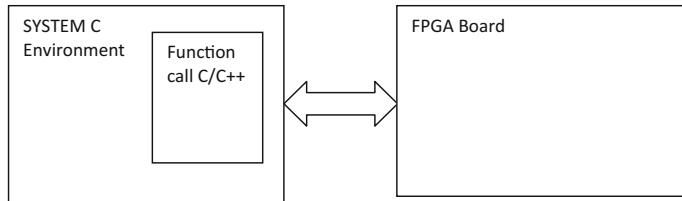
The following can be two important approaches to verify the million gate SOC.

### ***16.6.1 Hardware–Software Coverification***

In this, the signal-level bidirectional link can be used from simulator to the FPGA prototype. By using the simulator, API calls the communication can be established to verify the design under test. This can be used for the early detection of the bugs by writing the automated testbench. But if more functions need to verify, then such type of verification approach slows down the overall speed.

The strategy is to have the hardware–software coverification by using the bidirectional cycle accurate link between the FPGA prototyping board and the simulator. This can be accomplished by having the synthesizable wrapper for the Verilog or VHDL and other wrapper for the software simulator. The synthesizable code can be controlled by the hardware wrappers, and the non-synthesizable code can be controlled by the simulator. In this, the design is controlled by the simulator testbench and the data from hardware and software wrapper can be given to the bidirectional PLI interface.

But we can say that this type of approach has limitation as the hardware works at high speed and simulator at low operating frequency so may not be viable for the real-time interface. If the monitoring of the more internal registers is the requirement, then this slows the performance.



**Fig. 16.9** Transactor calls using System C

### 16.6.2 *Transactors and Transaction-Level Modeling*

This is one of the approaches which is recommended for the hybrid prototyping, and in this, the transaction link is used between the simulator and FPGA prototype. This can be achieved by having the testbench at the system level.

This can be done by using the System-C environment by using the transactors at the host PC software and DUT in the hardware. If the bidirectional link is established between the virtual model and the FPGA board, then by passing the message from the transactor the communication can be established. This type of verification strategy results in faster verification as compared to the hardware-software coverification (Fig. 16.9).

## 16.7 SOC Prototyping Future

Over the past decade, we have witnessed the substantial growth in the VLSI areas. The EDA companies and the chip manufacturing houses have evolved the nanometer SOC designs. In such circumstances, the miniaturization era is at the verge of evolution of the new trends and technological changes.

1. **Artificial intelligence:** The need of this century is the innovation of the products using the artificial intelligence mechanisms. The SOCs can be designed by embedding the artificial intelligence and algorithms. As the process node is shrinking further, there may be evolution like embedding the intelligence in the SOC. The SOCs can be used in the general purpose applications or in any kind of the sophisticated system, and the intelligence embedded systems can be controlled and configured using the reprogrammability using the complex FPGAs.
2. **Quantum computing:** The algorithms using quantum mechanism to improve the design speeds will be evolved in the future. The prize of the quantum computing machines will drop exponentially in the next few decades. We will really witness the high-speed low-power ASIC designs and FPGA prototyping using the new evolved techniques.
3. **High-density FPGAs:** We will witness the FPGA evolution using the lower process node. The evolution can give us the intelligent and high-density FPGAs

having high-speed interconnect, neural network cores and capability, programmable intelligence. These FPGAs can be used to validate the SOC designs in the area of

1. High-resolution and high-speed video processing
2. Testing of the neural network algorithms
3. Medical imaging and diagnosis
4. Satellite communications
5. Artificial intelligence.

## 16.8 Important Takeaways and Further Discussions

As discussed in the chapter, the SOC can be tested at the board level using the EDA tools, logic analyzer, oscilloscopes. The following are the important points:

1. Test the single FPGA design first, and then test the multiple FPGA boards.
2. Check for the interconnectivity between the multiple FPGAs using read/write transactions.
3. Check for the clock gating checks and redundant logic or logic removal effect.
4. Use the ILA cores with the Xilinx core generators to debug the design.
5. The MUX-based probing can be used if probe points are limited.
6. Use the ChipScope Pro to probe multiple signals at a time.
7. Check for the IO pin mapping, locations, standards, and voltage levels during debugging the prototype.
8. Check for the pin multiplexing data rate while debugging the design.
9. Use the hardware-software coverification to verify the complex designs.
10. Use the system C-based transactors to establish communication between the complex FPGA and the host.

## Reference

1. [www.xilinx.com](http://www.xilinx.com)

# Appendix A

## Few Synopsys Commands [1]

Few of the commands by Synopsys are listed below. They can be used during synthesis and timing analysis

Command	Description
<code>read-format &lt;format_type&gt; &lt;filename&gt;</code>	<i>Read the design</i>
<code>analyze -format &lt;format_type&gt; &lt;list of file names&gt;</code>	<i>Analyze the design for the syntax errors and translation before building the generic logic</i>
<code>elaborate -format &lt;list of module names&gt;</code>	<i>Used to elaborate the design</i>
<code>check_design</code>	<i>To check the design problems like shorts, open, multiple connection, instantiations with no connections</i>
<code>create_clock -name &lt;clock_name&gt; - period &lt;clock_period&gt; &lt;clock_pin_names&gt;</code>	<i>Create the clock for the design</i>
<code>set_clock_skew -rise_delay &lt;rising_clock_skew&gt; &lt;clock_name&gt;</code>	<i>Define the clock skew for the design</i>
<code>set_input_delay -clock &lt;clock_names&gt; &lt;input_delay&gt; &lt;input_port&gt;</code>	<i>To set the input port delay</i>
<code>set_output_delay -clock &lt;clock_names&gt; &lt;output_delay&gt; &lt;output_port&gt;</code>	<i>To set the output port delay</i>
<code>compile -map_effort &lt;map_effort_level&gt;</code>	<i>To compile with the low, medium or high map effort level</i>
<code>write -format &lt;format_type&gt; output &lt;file_name&gt;</code>	<i>To save the output generated by the synthesis tool</i>
<code>set_false_path -from [get_ports &lt;port list&gt;] -to get_ports &lt;port list&gt;</code>	<i>To set the false path</i>
<code>set_multicycle_path -setup &lt;period&gt; -from [get_cells] -to [get_cells]</code>	<i>To push the setup for the design having multicycle path</i>

(continued)

(continued)

Command	Description
<code>set_multicycle_path -hold &lt;period&gt; -from [get_cells] -to [get_cells]</code>	<i>To push the hold for the design having multicycle path</i>
<code>set_clock_uncertainty</code>	<i>To define the estimated network skew</i>
<code>set_clock_latency</code>	<i>To define the estimated source and network latency</i>
<code>set_clock_transition</code>	<i>Define the estimated clock skew</i>
<code>set_dont_touch</code>	<i>Used to prevent the optimization of the mapped gates</i>

For more information please visit

1. <http://www.synopsys.com/>

# Appendix B

## XILINX-7 Series Family

The Xilinx-7 series family comparison and the resource summary are listed below. For more information visit [www.xilinx.com](http://www.xilinx.com).

- **XILINX-7 series family comparison**

Max. capability	Spartan-7	Artix-7	Kintex-7	Virtex-7
Logic calls (K)	102	215	478	1955
Block RAM <sup>a</sup> (Mb)	4.2	13	34	68
DSP slices	160	740	1920	3600
DSP performance <sup>b</sup> (GMAC/s)	176	929	2845	5335
Transceivers	–	16	32	96
Transceiver speed	–	6.6 Gb/s	12.5 Gb/s	28.05 Gb/s
Serial bandwidth	–	211 Gb/s	800 Gb/s	2784 Gb/s
PCIe interface	–	x4 Gen2	xB Gen2	x8 Gen3
Memory interface (Mb/s)	800	1066	1866	1866
I/O pins	400	500	500	1200
I/O voltage (V)	1.2–3.3	1.2–3.3	1.2–3.3	1.2–33
Package options	Low-cost, wire-bond	Low-cost, wire-bond, lidless flip-chip	Lidless flip-chip and high-performance flip-chip	Highest performance flip-chip

<sup>a</sup>Additional memory available in the form of distributed RAM

<sup>b</sup>Peak DSP performance numbers are based on symmetrical filter Implementation

- **Virtex-7 FPGA feature summary**

Device <sup>a</sup>	Logic cells	Configurable logic blocks (CLBs)	DSP slices <sup>c</sup>	Block RAM blocks <sup>d</sup>		CMOS <sup>e</sup>	PCIe <sup>f</sup>	GTX	GTH	GTZ	XADC blocks	Total I/O BANKS <sup>g</sup>	Max user I/O <sup>h</sup>	SLRs <sup>i</sup>		
				Slices <sup>b</sup>	Max distributed RAM (KB)											
XCTV585T	582,720	91,050	69,38	1260	1590	795	28,620	18	3	36	0	0	17	850	N/A	
XCTV2000T	1,954,560	305,400	21,550	2160	2584	1292	46,512	24	4	36	0	0	1	24	1200	4
XCTVX330T	326,400	51,000	4388	11,20	1500	750	27,000	14	2	28	0	1	14	700	N/A	
XCTVX415T	412,160	64,400	6525	2160	1760	880	31,680	12	2	48	0	1	12	600	N/A	
XCTVX485T	485,760	75,900	8175	2800	2060	1030	37,080	14	4	56	0	0	1	14	700	N/A
XCTVX550T	554,240	86,600	8725	2880	2360	1180	42,380	20	2	80	0	1	16	600	N/A	
XCTVX690T	693,120	108,300	10,888	3600	2940	1470	52,320	20	3	80	0	1	20	1000	N/A	
XCTVX990T	979,200	153,000	13,838	3600	3000	1500	54,000	18	3	72	0	1	18	300	N/A	
XCTVX1140T	1,139,200	178,000	17,700	3360	3760	1880	67,880	24	4	96	0	1	22	1100	4	
XCTVH580T	580,480	90,700	8850	1680	1880	940	33,340	12	2	48	8	1	12	600	2	
XCTVH870T	876,160	136,900	13,275	2520	2820	1410	50,760	18	3	0	72	16	1	6	300	3

<sup>a</sup>EasyPath™-7 FPGAs are also available to provide a fast, simple, and risk-free solution for cost reducing Virtex-7 T and Virtex-7 XT FPGAs designs.

<sup>b</sup>Each 7 series FPGA slice contain four LUTs and eight flip-flops; only some slices can use their LUTs as distributed RAM or SRAMs.

<sup>c</sup>Each DSP slice contains a preadder, a  $25 \times 18$  multiplier, an adder, and an accumulator.

<sup>d</sup>Block RAMs are fundamentally 36 Kb in size; each block can also be used as two independent 18 Kb blocks.

<sup>e</sup>Each CMT contains one MMGM and one PLL.

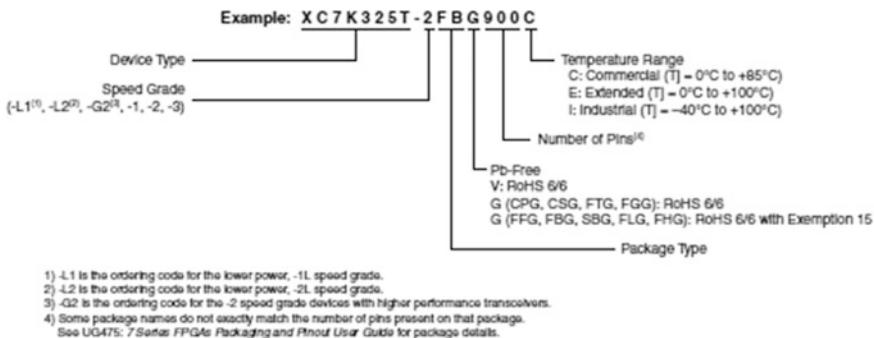
<sup>f</sup>Virtex-7 T FPGA Interface Blocks for PCI Express support up to x8 Gen 2, Virtex-7 XT and Virtex-7 HT Interface Hocks for PCI Express support up to xB Gen 3, with the exception of the XC7VX485T device, when supports X8 Gen 2.

<sup>g</sup>Does not include configuration Bank 0.

<sup>h</sup>This number does not include GTX, GTH, or GTZ transceivers.

<sup>i</sup>Super logic regions (SLRs) are the constituent parts of FPGAs that use SSI technology to connect SLRs with 28.65 Gb/s transceivers.

- **Xilinx-7, Artix-7, Kintex-7, Virtex-7 ordering information**



DS180\_01\_061317

For more information please use the following link

1. <http://www.xilinx.com/>

## Appendix C

### Intel FPGA Stratix 10 Devices

The Intel FPGA Stratix 10 series family comparison and the resource summary are listed below. For more information visit [www.altera.com](http://www.altera.com)

- **Intel FPGA Stratix 10 core plan**

Intel Stralix 10 GX/SX device name	Logic elements (KLE)	M20K blocks	M20K Mbits	MLAB counts	MLAB Mbits	$18 \times 19$ multipliers <sup>a</sup>
GX 400/SX 400	378	1537	30	3204	2	1296
GX 650/SX 650	612	2469	49	5184	3	2304
GX 850/SX 850	841	3477	68	7124	4	4032
GX 1100/SX 1100	1092	4401	86	9540	6	5040
GX 1650/SX 1650	1624	5851	114	13,764	8	6290
GX 2100/SX 2100	2005	6501	127	17,316	11	7488
GX 2500/SX 2500	2422	9963	195	20,529	13	10,022
GX 2800/SX 2800	2753	11,721	229	23,796	15	11,520
GX 4500/SX 4500	4463	7033	137	37,821	23	3960
GX 5500/SX 5500	5510	7033	137	47,700	29	3960

- **Interconnect, PLL, and hard IPs FPGA core**

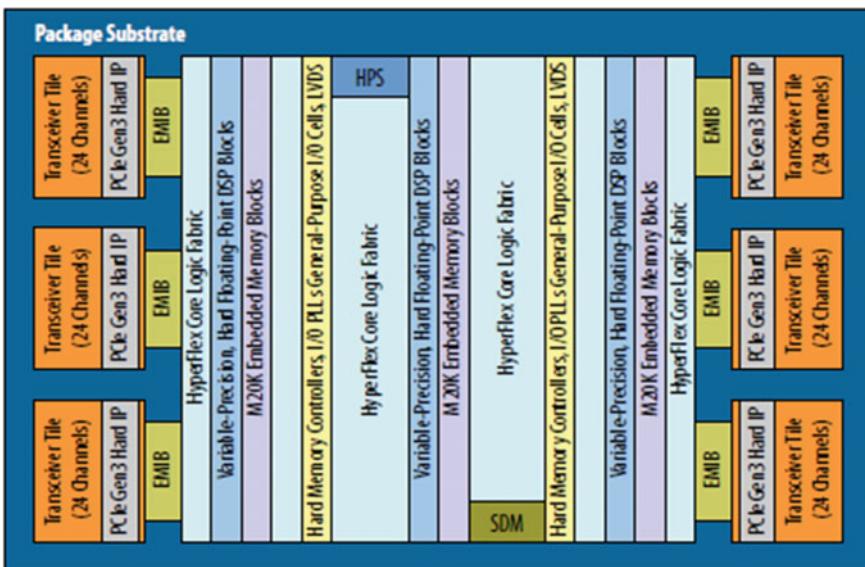
Intel Stratix 10 GX/SX device name	Interconnnects		PLLs		Hard IP PCIe hard IP blocks
	Maximum GPIOs	Maximum XCVR	fPLLs	I/O PLLs	
GX 400/SX 400	392	24	8	8	1
GX 650/SX 650	400	48	16	8	2
GX 850/SX 850	736	48	16	15	2
GX 1100/SX 1100	736	48	16	15	2
GX 1650/SX 1650	704	96	32	14	4

(continued)

(continued)

Intel Stratix 10 GX/SX device name	Interconnects		PLLs		Hard IP
	Maximum GPIOs	Maximum XCSR	fPLLs	I/O PLLs	
GX 2100/SX 2100	704	96	32	14	4
GX 2500/SX 2500	1160	96	32	24	4
GX 2800/SX 2800	1160	96	32	24	4
GX 4500/SX 4500	1640	24	8	34	1
GX 5500/SX 5500	1640	24	8	34	1

- Intel FPGA Stratix 10 family architecture

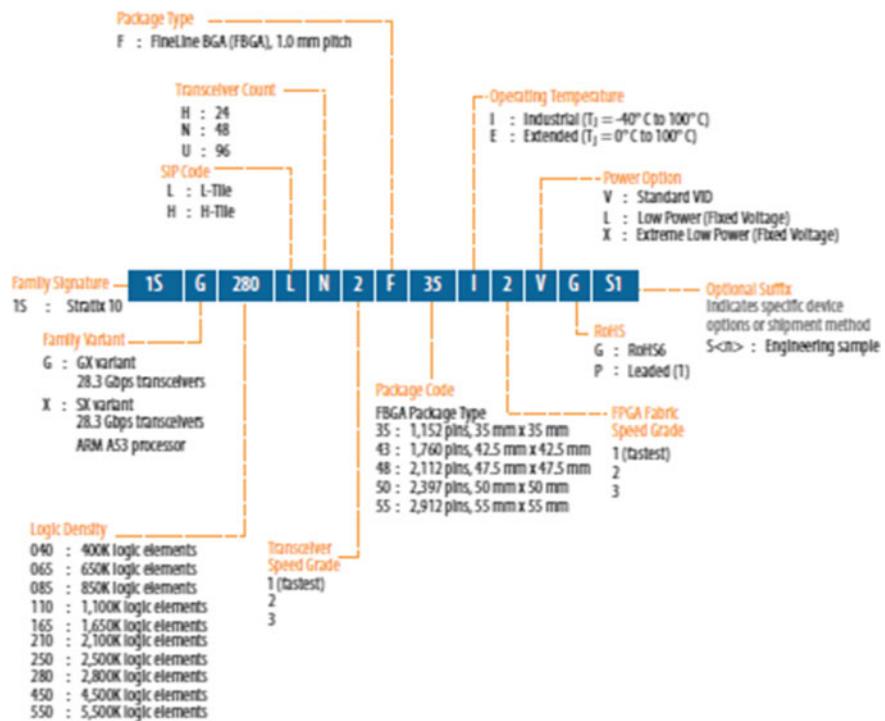


HPS: Quad ARM Cortex-A53 Hard Processor System

SDM: Secure Device Manager

EMIB: Embedded Multi-Die Interconnect Bridges

## • Intel Stratix 10 FPGA family sample ordering information



Note:

1. Contact Intel for availability

For more information please use the following link

1. <http://www.altera.com>

# Index

## A

Accelerator Coherence Port (ACP), 272  
Active clock edge, 174  
ADC, 143, 160  
ADC and DAC logic, 221  
Adders, 42, 143  
Address bus, 64  
AES encryption key, 226  
AHB and APB bus, 114, 134  
AHB/APB, 114  
ALM, 261  
Analog, 9  
Analog blocks, 221, 264  
Analog IPs, 159  
Analysis modes, 188  
API, 288  
A place and route tool uses, 167  
Application Processing Unit (APU), 271  
Application Specific Integrated Circuit (ASIC),  
    4  
Application Specific Standard Product (ASSP),  
    5  
AP-SOC, 270  
Architecting the SOC, 67  
Architecture, 8, 19, 61, 165, 189, 235, 267  
Architecture/design changes, 267  
Area, 8, 21, 160  
Area report, 244  
ARM clock, 275  
ARM IP, 209  
ARM processor, 270  
ARM processor system, 114  
Artificial intelligence, 15  
ASIC clock gating, 44

ASIC library, 161, 234  
ASIC memories, 239  
ASIC testing, 191  
ASIC/FPGA designs, 170  
ASIC/SOC designs, 159  
A. S. Rock, 2  
Asynchronous multiplexing, 226  
Audio processing, 8  
Automatic, 239  
Automatic partitioning, 233, 239  
AXI ports, 272

## B

Backend flow, 170, 242  
Backend tool, 166, 265  
Bandwidth, 254  
Base array, 10  
Basic tests, 264  
48-bit accumulator, 149  
64 bit ECC, 135, 204  
Bit-stream, 287  
Blocking assignments, 25, 27  
Block level constraints, 164  
Block RAM (BRAM), 131, 135, 169, 204, 218,  
    238, 265, 273, 286  
Board layout, 251  
Bottom up approach, 163  
Boundary scan, 234  
Bus bandwidth, the, 64  
Bus clock, 275  
Buses, 65  
Bus logic, 8  
By-passable registers, 261

- C**
- Cable delay, 254
  - Cache controller, 66
  - Cache memory, 65
  - Capture FPGA, 248
  - Cascade, 33
  - Cascaded output adder chain, 150
  - Case, 50
  - Cell layout, 10
  - Cell library, 2, 136, 204
  - Central crossbar, 272
  - Certify Pin Multiplier (CPM), 228
  - Channeled gate array, 10
  - Channel less gate array, 10
  - ChipScope pro, 286
  - Circuit, 248
  - Circular buffers, 249
  - CLB, 169, 194, 265
  - Clean timing paths, 159
  - Clock and reset network, 233
  - Clock buffers, 170
  - Clock distribution, 225
  - Clock divider, 191
  - Clock enable, 45, 50
  - Clock gating, 170, 192
  - Clock gating cells, 44, 170
  - Clock gating logic, 234
  - Clock generators, 266
  - Clocking architecture, 207
  - Clock Management Tiles (CMT), 202
  - Clock multiplexing, 191
  - Clock network delay, 235
  - Clock network latency, 190
  - Clock path, 187
  - Clock port, 180
  - Clock rate, 7, 64
  - Clock skew, 170, 177, 269
  - Clock skew distribution, 266
  - Clock spine, 207
  - Clock tree, 225
  - Clock tree synthesis, 8, 21
  - Combinational, 184
  - Combinational loops, 26
  - Combinational partitioning, 223
  - Combinational paths, 214
  - Compatible interfaces, 264
  - Complex functionality, 200
  - Computational elements, 143
  - Concurrent execution, 64
  - Concurrent IO data transfer, 67
  - Configuration, 279
  - Congestion, 268
  - Constraint file, 184
  - Constraints, 19, 61, 161, 268, 283
  - Constraints (SDC), 179
  - Content addressable, 135, 204
  - Controllability, 23
  - Core generators, 168
  - Cortex A9, 271
  - Coverage goals, 264
  - Co-verification, 289
  - Coverification and use of IPs, 23
  - Critical path, 195
  - Critical path delay, 261
  - Crosstalk, 248
  - Custom interfaces, 23
  - Cycle based, 288
- D**
- DAC, 143, 160
  - Data arrival, 182
  - Data Arrival Time (AT), 182, 187
  - Data bus, 64
  - Data convergence, 42
  - Data input, 180
  - Data integrity, 42, 177
  - Data memories, 66
  - Data path, 189
  - Data rate, 67, 160
  - Data Required Time (RT), 182
  - DDR3, 275
  - DDR4 memory interfaces, 137
  - Debug, 23
  - Debug Access Port (DAP), 272
  - Debug and test circuitry, 252
  - Debug plan, 278
  - Decrypted, 226
  - Derating, 187
  - Design compiler, the, 200
  - Design constraints, 21, 161
  - Design library, 200
  - Design partitioning, 144, 166, 199, 252
  - Design performance, 66
  - Design planning, 278
  - DesignWare, 161
  - DesignWare library, 162
  - DFT, 13
  - Differential IO pairs, 265
  - Differential signal, 258
  - Differential signals between, 252
  - Digital, 9
  - Digital blocks, 221
  - Digital Signal Processing (DSP), 9, 202
  - Direct interface, 248
  - Direct port connections, 69
  - Distributed RAM, 120, 169
  - DMA controller, 272
  - DMA interface, 144

- Don't touch, 166, 234  
Double Data Rate (DDR), 137  
DRAM, 275  
Drive strengths, 265  
DSP algorithm, 141, 143, 149, 169, 206  
DSP block, 142, 150, 160, 169, 218, 238, 266, 267  
DSP48E1, 206  
DSP48E1 slice, 149  
DSP processor, 141, 143  
DSP slices, 273  
Dual port, 135, 204  
Dual port RAM, 120, 121, 169  
Dynamic, 177  
Dynamic connectivity, 251  
Dynamic power, 50, 170  
Dynamic simulator, 177  
Dynamic Timing Analysis (DTA), 177
- E**
- EDA tools, 252  
Efficiency, 260  
Embedded systems, 64  
Emulation, 22  
Encrypted key, 226  
Encrypted source code, 226  
Endpoint, 180  
Energy density, 66  
Error corrections, 137, 205  
Ethernet, 266  
Ethernet MAC, 209  
Event based simulator, 288  
Extensible Processing Platform (EPP), 270  
External interfaces, 267  
External IP board, 265  
External memory, 134, 264  
Extracted nets (SPEF), 179
- F**
- False path, 179, 191  
Fan-out, 268  
FFT algorithms, 152  
FIFO, 132, 135, 160, 204, 249  
FIFO or circular buffers, 144  
Filtering, 265  
FLG1925, 250  
Floating point mode, 151  
Floating point operations, 144  
Floorplan, 8  
Floorplanning, 13  
Floor planning tool, 168  
Formal Verification (FV), 217  
FPGA bit-stream file, 167  
FPGA boundary, 283
- FPGA clock gating, 44  
FPGA connectivity, 269  
FPGA editors, 168  
FPGA-equivalent, 149  
FPGA fabric, 136, 171, 204, 209, 269, 270  
FPGA functional, 194  
FPGA gate count, 266  
FPGA interfaces, 22  
FPGA netlist, 166, 264  
FPGA pad, 259  
FPGA performance, 268  
FPGA resources, 233, 244, 264, 266  
Frame prediction logic, 148  
Frame processing logic, 148  
FSM controller's, 160  
FSMs, 26, 132  
Full custom, 7  
Functional and timing proven IPs, 166  
Functional correctness, 264  
Functional coverage, 288  
Functional verification, 21, 264, 278
- G**
- Gate array, 9  
Gated clock implementation, 170  
Gate level netlist, 161, 179  
Gating strategy, 44  
GDSII, 21  
General purpose IO, 8, 137  
General purpose registers, 68  
Generated clocks, 191  
Gigabit transceivers, 270  
Glitches, 26, 44  
Global clock, 200  
Global Foundries (GF), 6  
Glue logic, 234  
Gordon Moore, 1  
Grouping, 32, 234  
GTECH, 226
- H**
- H.264 encoder, 148  
Hamming code, 137, 205  
Hard floating point adders, 150  
Hard IP, 226, 264  
Hard macros, 266  
Hard or soft memory controllers, 137  
Hard processor DSP core, 143  
Hardware software, 289  
Hardware and software partitioning, 159  
HDL synthesis, 234  
Hierarchical designs, 4, 234  
High impedance, 34, 251  
High performance IOB, 207

High precision fixed point, 150  
 High speed interfaces, 111  
 High speed IO, 160, 218, 233, 240  
 High Speed Time Division Multiplexing (HSTDMD), 228  
 Hold analysis, 188  
 Hold check, 183  
 Hold slack, 183, 193  
 Hold time violation, 188  
 Hybrid Memory Cube (HMC), 138  
 Hyperflex core architecture, 260  
 Hyper registers, 261  
 Hyper scaling, 2

**I**

IEEE standard 802.3-2005, 209  
 If else, 50, 54  
 ILA blocks, 287  
 Implementation tool, 166  
 In circuit debugging tool, 168  
 Incremental synthesis, 241  
 Inference, 234  
 Input port, 180  
 Input to output, 178  
 Input to output path, 181  
 Input to register, 178  
 Input to register Path, 180  
 Input vectors, 177  
 Instantiation, 234  
 Instruction execution, 68  
 Integrated Logic Analyzer (ILA), 286  
 Intel, 6  
 Intel-FPGA, 200, 260  
 Intel-FPGA transceiver, 111  
 Intel's Nios, 138  
 Intel Stratix 10, 135  
 Interconnect, 4  
 Interconnect delay, 224, 248, 268  
 Interconnect issues, 235  
 Interconnectivity, 235  
 Interconnect networks, 265  
 Interface, 20, 278  
 Interface delays, 265  
 Interface timing, 160  
 Inter-FPGA communication, 280  
 Internal memory buffers, 148  
 International Technology Roadmap for Semiconductors (ITRS), 3  
 In the quantum computing, 15  
 IO availability, 160  
 IO bandwidth, 67  
 I/O bank, 137  
 IO blocks, 223  
 IO cell, 259

IO columns, 207  
 IO data rate, 144  
 IO delay, 194, 228, 269  
 IO devices, 116  
 IO interfaces, 64, 250  
 IO multiplexing, 255, 267  
 IO pad, 259  
 IO pad instance, 259  
 IO pad rings, 264  
 IO pin count, 248, 266  
 IO pins, 200, 258, 265  
 IO placement, 168  
 IOs, 267  
 IO speed, 254  
 IO tests, 278  
 IO-SERDES, 228  
 IP, 8, 198, 217, 254  
 IP blocks, 264  
 IP cores, 168, 234  
 IP size, 260

**J**

JTAG, 286  
 JTAG controller, 287

**L**

Late clock, 187  
 Late derating, 187  
 Latency, 160, 249  
 Launch FPGA, 248  
 LC tank circuit, 107, 208  
 Libraries (.lib), 21, 161, 179  
 Linear Feedback Shift Register (LFSR), 144  
 Link library, 162  
 Local instruction, 66  
 Logical partitioning, 233  
 Logic analyzer, 278, 279, 283  
 Logic density, 265  
 Logic replication, 235  
 Logic replications and resource sharing, 218  
 Low power gigabit transceiver, 107, 208  
 Low Voltage Differential Signals (LVDS), 134, 139, 223, 228, 258  
 LUT delay, 194, 214  
 LUTs, 136, 169, 204, 267

**M**

MAC, 143, 265  
 Macros, 10  
 Macros, RAMs, 217  
 Manual, 239  
 Mapping, 235  
 Maximum transfer frequency, 283  
 Mealy machine, 26

- Memories, 267  
Memory controller, 134  
Memory generator, 136, 204  
Metastability, 175, 176  
Metastable, 174  
Metastable state, 42  
Micro-architecture, 8, 61, 73, 189, 235  
Microprocessor core, 66  
Minimum delay, 187  
Mixed interconnects, 250  
Mixed signal designs, 178  
Moore machines, 26  
Moore's second law, 2  
Multicycle paths, 192  
Multimedia, 8  
Multiple clock domain design, 176  
Multiple clock domains, 200, 218  
Multiple clocks, 42  
Multiple FPGA, 170, 264, 266–268, 278  
Multiple FPGA designs, 221  
Multiple pipelined stage, 268  
Multiple processors, 8  
Multiple state machines, 148  
Multiplexed bus, 64  
Multiplexed IO, 269, 280  
Multiplexers, 42  
Multiplexing, 227  
Multipliers, 143, 267  
Multiply and accumulate, 150, 169, 206  
Multitasking, 19, 67, 143  
MUX and Demux, 255  
MUX based logic, 34
- N**  
Nanometer, 2, 198  
NBA, 54  
Need of multiple FPGA, 22  
Netlist, 233  
Non blocking assignments, 25  
Non-Return-to-Zero (NRZ), 110  
Non vectored, 177  
Non-vectored approach, 178  
NRE cost, 3
- O**  
Observability, 23  
Off-FPGA, 252  
On-board delay, 195, 224, 228, 252  
On-chip delay, 195, 252  
One-hot encoding, 160  
One-Time Programmable (OTP), 201  
On-FPGA, 252  
Operating conditions, 187  
Operating frequency, 143, 194  
Optimization, 21, 159, 160, 241  
Oscillatory behavior, 216  
Oscilloscope, 285  
Output port, 180  
Overall timing, 261
- P**  
Package, 66  
Pad library, 259  
P and R tool, 241, 278  
Parallelism, 143  
Parallel logic, 33  
Parallel to serial converter, 107, 208  
Partition, 218  
Partitioned, 280  
Partitioned design, 193, 267  
Partitioning, 22, 170, 239  
Partitioning tool, 222, 239  
Partition the design, 222  
Pattern detector, 206  
PCI express (PCIe), 202, 209, 260, 266  
Performance, 233, 248  
Performance improvement, 160  
Phy, 209  
Physical design, 4  
Physical IP, 226  
Pin count, 66, 258  
Ping-pong buffers, 148  
Pin multiplexing, 226, 280  
Pin multiplexing need, 248  
Pin placements, 166  
Pin requirements, 240  
Pipelined, 266  
Pipelined registers, 214  
Pipelining, 92, 268  
Place and Route (PR), 21, 217, 229, 265  
Placement, 8  
Placement and routing, 168, 268  
Placement and routing tools, 235  
PLL, 107, 190, 208, 225, 266  
Positive slack, 187, 188  
Post layout verification, 21  
Post-synthesis, 28  
Power, 8, 21, 41, 66, 160, 266  
Power domains, 160  
Powerplan, 8  
Power planning, 21  
Pre-adders, 150  
Pre-synthesized netlist, 226  
Priority encoders, 54  
Probe points, 240  
Processor architecture, 64, 144  
Processor cores, 64, 266  
Processors, 159

- Process, voltage, or temperature, 138  
 Programmable switches, 251  
 Program memory and data memory, 144  
 Propagation delays, 268  
 Prototype and testing, 264  
 Prototype boards, 260  
 Prototype flow, 264  
 Prototyping boards, 264  
 Prototyping flow, 241  
 PVT variations, 134
- Q**
- Queues, 67
- R**
- RAM, ROM, FIFO, 265  
 Real time processing, 143  
 Reconfigurable memory, 133  
 Register balancing, 92  
 Registered inputs, 160  
 Registered outputs, 160  
 Register optimization, 92  
 Register to output path, 178, 180  
 Register to register path, 178, 180  
 Required time, 182  
 Requirement analysis, 10  
 Reset logic, 234  
 Resource sharing, 39, 42  
 Resource utilization, 267  
 Ring oscillators, 107, 208  
 Ring type arrangement, 250  
 Robots, 15  
 Rock's law, 2  
 ROM, 135, 204  
 Routing, 8, 66  
 Routing delay, 261, 268  
 RTL, 259  
 RTL design, 21, 67, 144, 160, 264  
 RTL source code of IP, 226  
 RTL tweaks, 228  
 RTL verification, 264
- S**
- Sample and hold, 143  
 Sampling frequency, 143  
 Samsung, 6  
 Scan insertion, 13  
 SDF, 178  
 SDRAM controller, 127  
 Semiconductor, 3  
 Semi custom, 7  
 Sensitivity list, 216  
 SERDES, 255, 258, 266  
 Serial to parallel converter, 110, 209
- Serial transceiver, 107, 208  
 Settling time, 269  
 Setup and hold check, 192  
 Setup slack, 183, 193  
 Setup time, 181  
 Shared bus, 65  
 Shifter, 143, 255  
 Signal integrity, 235, 248, 264, 269, 283  
 Signed multiplier, 206  
 Silicon complexity, 4  
 Silicon wafer, 9  
 Simulation, 21  
 Simulator, 288  
 Single port, 135, 204  
 Single port RAM, 121  
 Single precision floating point, 150  
 Six input logic function, 169  
 Skew, 225  
 Skew, slow transition, 285  
 Slack, 182  
 SLICEL, SLICEM, 203  
 SOC architecture, 250, 266  
 SOC prototype, 214  
 SOC prototyping, 194  
 SOC speed, 241  
 Soft and hard memory controller cores, 134  
 Soft cores, 134  
 Spartan 3, 132  
 Special low power, 110, 209  
 Specifications, 19  
 Speech synthesis, 15  
 Speed, 8, 21, 160, 268  
 Speed or performance, 166  
 Speed, power and bandwidth, 143  
 SRAM, 123, 200  
 SRAM controller, 123  
 Stacked Silicon Interconnects (SSI), 202  
 Standard cell library, 10, 163  
 Standard cells, 10  
 Standard IO, 272  
 Standard protocols, 111  
 Star topology, 250  
 Start point, 180  
 Static, 177  
 Static RAM, 131  
 Static Timing Analysis (STA), 21, 177  
 Stratix 10, 260  
 Structured gate array, 10  
 Switched routing matrix, 251  
 Switch matrix, 251  
 Symbol libraries, 163  
 Synchronization, 166, 234  
 Synchronizer, 160, 175, 176, 218, 234  
 Synchronous, 132

- Synchronous and asynchronous reset, 265  
Synchronous circuit, 173  
Synchronous multiplexing, 226  
Synopsys Design Compiler (SDC), 161, 259  
Synopsysfull case, 28  
Synopsysparallel case, 30  
Synthesis, 21, 233, 241  
Synthesis optimization, 233  
Synthesis results, 267  
Synthesis tool, 171, 268  
Synthesis tool directives, 264  
Synthesizable model, 259  
System clock, 258
- T**  
Taiwan Semiconductor Manufacturing Corporation (TSMC), 6  
Test Access Port (TAP), 209  
Target library, 162  
TDM, 223, 280  
TDM and LVDS, 218  
Technology library, 161, 217, 234  
Termination impedance, 248  
Test and debug logic, 267  
Testbench, 21, 289  
Test coverage, 177  
Testing, 4  
Test insertion, 12  
Throughput, 11, 68, 144, 260  
Timed paths, 178  
Time to market, 3, 178  
Timing, 20, 238  
Timing analyzer, 173  
Timing closure, the, 178  
Timing constraints, 174, 178  
Timing estimations, 252  
Timing margin, 138  
Timing paths, 173  
Timing performance, 178, 268  
Timing report, 184, 189, 244  
Timing sequence, 175  
Timing violation, 178, 187  
Top level boundary, 259  
Top level constraints, 166  
Top module, 61  
Transaction level modelling, 288  
Transactors, 289  
Transceivers, 202
- Transmission line, 283  
Tri state, 23  
Tri-state logic, 34  
Two's-complement multiplier, 149
- U**  
Unconstrained path, 184  
Uniform clock latency, 225  
Uniform clock skew, 225  
Uniform skew, 233  
Utilization, 254
- V**  
Variable precision DSP block, 152  
Vectors, 177  
Vehicle controls, 15  
Verification, 4, 233  
Verilog or VHDL, 288  
Video, 8  
Video decoders, 160  
Video decoding system, 148  
Video encoders and the decoders, 148  
Video encoding system, 148  
Virtex, 202  
Virtex 7, 224  
Virtex7 (XC7V2000T), 250, 265  
Virtex7 7VX200T, 267  
Virtex XC7V200T, 265  
Voltage and logic levels, 265  
Voltage domains, 160
- W**  
Weak driving strength, 285  
Wire delays, 225  
Wrapper file, 217  
Write and read transactions, 114
- X**  
Xilinx, 199, 200  
Xilinx and Intel, 107  
Xilinx and Intel FPGAs, 141  
Xilinx core generators, 217  
Xilinx 7 series, 135, 149, 204, 270
- Z**  
Zynq 7000, 270  
Zynq boards, 266