

Vaibbhav Taraate

# Digital Logic Design Using Verilog

Coding and RTL Synthesis

 Springer

# Digital Logic Design Using Verilog



Vaibbhav Taraate

# Digital Logic Design Using Verilog

Coding and RTL Synthesis

Vaibhav Taraate  
Semiconductor Training @ Rs.1  
Pune, Maharashtra  
India

ISBN 978-81-322-2789-2      ISBN 978-81-322-2791-5 (eBook)  
DOI 10.1007/978-81-322-2791-5

Library of Congress Control Number: 2016936278

© Springer India 2016

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.

Printed on acid-free paper

This Springer imprint is published by Springer Nature  
The registered company is Springer (India) Pvt. Ltd.

*Dedicated To the Supreme Lord  
of Intelligence*

*And*

*To All Students and Readers!*

# Preface

Today's century is era of miniaturization and high-speed chips, and complex ASICs are designed in lesser time as compared to before. The technology evolution from 1990 has opened up a new paradigm for ASIC designers. Customers are always expecting the speedy delivery of the ASIC products and it always accumulates the good amount of pressure to come up with the high-performance design using less number of resources.

The evolution in the process node technology in the past decade has started the real evolution in the semiconductor industry! Many new design techniques and flows got evolved and stabilized in the past decade. Many EDA tool companies help designers to complete the design in shorter time span.

In today's industrial scenario, designer doesn't spend more time to draw the schematic to design the digital logic circuits. The EDA tools have enabled the best design practices by using hardware description languages such as VHDL and Verilog. The synthesis tools are used primarily to convert the HDL into the equivalent logic structure or gate level netlist. The latest EDA tool features have also improved the productivity and efficiency of the design team!

The book is organized into three sections; the first section consists of Chaps. 1–9 and describes about the digital logic design and synthesizable Verilog RTL. Section I is organized in such a way that reader will be able to have better understanding of basics of digital logic and synthesizable RTL. This section will be helpful for the reader to understand the Verilog HDL constructs, hardware inference, simulation concepts and design guidelines for simple to complex designs. For the better understanding of the reader, few practical scenarios are included in this section.

Chapter 1 discusses about the evolution of the logic design, logic design abstraction levels, IC design methodologies and flow, Verilog Module declaration, and different design styles. This chapter discusses about the simulation and synthesis flow for the Verilog RTL. Even this chapter discusses about the key verilog HDL features.

Chapters 2 and 3 describe about the combinational logic design and synthesizable RTL. These chapters also focus on the practical issues and scenarios while designing the combinational logic using Verilog RTL.

Chapter 4 discusses on the key Verilog coding guidelines and the role of Verilog in writing an efficient RTL for combinational design.

Chapter 5 discusses about the sequential logic design and covers most of the simple to complex practical design scenarios. This chapter also deals with the synthesizable sequential design issues, timing diagrams, and simulation of the design.

Chapter 6 discusses on the key Verilog coding guidelines and the role of Verilog in writing an efficient RTL for sequential design.

Chapter 7 describes about the efficient RTL coding for a few complex density designs and also gives information about the synthesizable results and the key practical scenarios for the design.

Chapter 8 deals with FSM and design of an efficient FSM using the suitable FSM encoding styles.

Chapter 9 describes the simulation concepts and PLD based design. Even this chapter describes about the design guidelines while using PLDs.

Section II consists of Chaps. 10–12 and mainly deals with the logic synthesis, the static timing analysis, and the constraining ASIC designs. This section is organized in such a way that reader can have better understanding of synthesizable Verilog RTL and constraining designs for given specifications. This section also deals with the static timing analysis and practical issues in performance improvement for the design.

Chapter 10 discusses about the logic synthesis, ASIC design flow, design constraints, and gate level netlist.

Chapter 11 describes the static timing analysis and the timing reports and analysis for complex RTL designs. This chapter also deals with the practical few practical scenarios in the design and performance improvement technique.

Chapter 12 discusses about the different design constraints and how to tweak architecture, microarchitecture, and RTL to improve the design performance. This chapter also deals with the DRC, optimization and performance improvement scenarios for better understanding of the design constraints.

Section III consists of Chaps. 13–15 and mainly discusses on the advanced RTL design concepts such as multiple clock domain designs, need of synchronizers, clock domain crossing, low power designs, and SOC-based designs and challenges. Every chapter in this section discusses about the key practical scenarios using the efficient Verilog RTL.

Chapter 13 describes about the multiple clock domain designs and the synchronizers and their need. This chapter also focuses on synchronous and asynchronous FIFO buffers and RTL design using Verilog and concludes with a case study.

Chapter 14 discusses on most of the low power design techniques and the goal of designers to implement the low power designs. This chapter also deals with the low power design architecture and power sequencing for the low power designs.

Chapter 15 focuses on the real-life SOC-based designs and the role of Verilog in implementing the SOC-based designs.

The book consists of many practical examples from simple to complex logic depth. This will enable the reader to have better understanding about how to code an efficient RTL using Verilog. The synthesizable designs, and frequent issues in the RTL design life cycle are organized in each section for the better understanding.

This book is targeted to the engineering students, inexperienced engineers, and professionals those who want to implement practical, synthesizable, efficient RTL using Verilog!

# Acknowledgments

This book is possible due to help of many people. I truly appreciate their direct and indirect help during writing of this book. Among them I am very much thankful to my dearest friend, Ishita Thaker (Ish), for encouraging me to write this book. This book would not have been possible if my wife Somi has not reviewed the book contents and grammatical mistakes.

Special thanks to my son Siddesh and my daughter Kajal for their ideas and creative thoughts while creating diagrammatic representation for this book. I truly appreciate the sacrifices of Siddesh and Kajal.

Special thanks to all the students to whom I taught the subject for more than one decade. Indirectly I want to thank all my teachers for their valuable help during my engineering and postgraduation at IIT Powai (Mumbai).

Special thanks to all the Springer staff, especially Swati Maherishi and Aparajita Singh for good and encouraging conversations, support, and encouragement.

Special thanks in advance to all those readers across the world for buying, reading, and enjoying the book!

# Contents

<b>1</b>	<b>Introduction</b>	1
1.1	Evolution of Logic Design	1
1.2	System and Logic Design Abstractions	3
1.3	Integrated Circuit Design and Methodologies	4
1.3.1	RTL Design	4
1.3.2	Functional Verification	5
1.3.3	Synthesis	5
1.3.4	Physical Design	5
1.4	Verilog HDL	5
1.5	Verilog Design Description	7
1.5.1	Structural Design	7
1.5.2	Behavior Design	9
1.5.3	Synthesizable RTL Design	10
1.6	Key Verilog Terminologies	10
1.6.1	Verilog Arithmetic Operators	11
1.6.2	Verilog Logical Operators	11
1.6.3	Verilog Equality and Inequality Operators	11
1.6.4	Verilog Sign Operators	13
1.6.5	Verilog Bitwise Operators	16
1.6.6	Verilog Relational Operators	18
1.6.7	Verilog Concatenation and Replication Operators	18
1.6.8	Verilog Reduction Operators	19
1.6.9	Verilog Shift Operators	20
1.7	Summary	26
<b>2</b>	<b>Combinational Logic Design (Part I)</b>	27
2.1	Introduction to Combinational Logic	27
2.2	Logic Gates and Synthesizable RTL	28
2.2.1	NOT or Invert Logic	28
2.2.2	Two-Input OR Logic	28



2.2.3	Two-Input NOR Logic . . . . .	28
2.2.4	Two-Input AND Logic . . . . .	32
2.2.5	Two-Input NAND Logic . . . . .	33
2.2.6	Two-Input XOR Logic . . . . .	34
2.2.7	Two-Input XNOR Logic . . . . .	34
2.2.8	Tri-state Logic . . . . .	37
2.3	Arithmetic Circuits . . . . .	38
2.3.1	Adder . . . . .	38
2.3.2	Subtractor . . . . .	41
2.3.3	Multi-bit Adders and Subtractors . . . . .	44
2.3.4	Comparators and Parity Detectors . . . . .	46
2.3.5	Code Converters . . . . .	49
2.4	Summary . . . . .	51
<b>3</b>	<b>Combinational Logic Design (Part II)</b> . . . . .	<b>53</b>
3.1	Multiplexers . . . . .	53
3.1.1	Multiplexer as Universal Logic . . . . .	54
3.2	Decoders . . . . .	63
3.2.1	1 Line to 2 Decoder Using “case” . . . . .	63
3.2.2	1 Line to 2 Decoder with Enable Using “case” . . . . .	63
3.2.3	2 Line to 4 Decoder with Enable Using “case” . . . . .	63
3.2.4	2 Line to 4 Decoder with Active Low Enable Using ‘case’ . . . . .	67
3.2.5	4 Line to 16 Decoder Using 2:4 Decoder . . . . .	68
3.3	Encoders . . . . .	75
3.3.1	Priority Encoders . . . . .	77
3.4	Summary . . . . .	78
<b>4</b>	<b>Combinational Design Guidelines</b> . . . . .	<b>79</b>
4.1	Use of Blocking Assignments and Event Queue . . . . .	80
4.2	Incomplete Sensitivity List . . . . .	81
4.3	Continuous Versus Procedural Assignments . . . . .	82
4.4	Combinational Loops in Design . . . . .	85
4.5	Unintentional Latches in the Design . . . . .	88
4.6	Use of Blocking Assignments . . . . .	89
4.7	Use of If-Else Versus Case Statements . . . . .	91
4.8	MUX Nested or Priority Structure . . . . .	92
4.9	Decoder 2:4 . . . . .	92
4.10	Encoder 4:2 . . . . .	93
4.11	Missing ‘Default’ Clause in Case . . . . .	93
4.12	If-Else with Else Missing . . . . .	95
4.13	Logical Equality Versus Case Equality . . . . .	97
4.13.1	Logical Equality and Logical Inequality Operators . . . . .	97
4.13.2	Case Equality and Case Inequality Operators . . . . .	97

- 4.14 Arithmetic Resource Sharing . . . . . 98
  - 4.14.1 With Resource Sharing . . . . . 98
- 4.15 Multiple Driver Assignments . . . . . 102
- 4.16 Summary . . . . . 102
- 5 Sequential Logic Design . . . . . 103**
  - 5.1 Sequential Logic . . . . . 103
    - 5.1.1 Positive Level Sensitive D-Latch . . . . . 104
    - 5.1.2 Negative Level Sensitive D Latch. . . . . 106
  - 5.2 Flip-Flop. . . . . 107
    - 5.2.1 Positive Edge Triggered D Flip-Flop. . . . . 108
    - 5.2.2 Negative Edge Triggered D Flip-Flop . . . . . 108
  - 5.3 Synchronous and Asynchronous Reset . . . . . 109
    - 5.3.1 D Flip-Flop Asynchronous Reset . . . . . 109
    - 5.3.2 D Flip-Flop Synchronous Reset . . . . . 111
    - 5.3.3 Flip-Flop with Load Enable Asynchronous Reset . . . 112
    - 5.3.4 Flip-Flop with Synchronous Load and Synchronous Reset . . . . . 112
  - 5.4 Synchronous Counters . . . . . 114
    - 5.4.1 Three Bit Up Counter . . . . . 114
    - 5.4.2 Three-Bit Down Counter . . . . . 118
    - 5.4.3 Three-Bit Up-Down Counter . . . . . 120
    - 5.4.4 Gray Counters . . . . . 121
    - 5.4.5 Gray and Binary Counter. . . . . 123
    - 5.4.6 Ring Counters . . . . . 125
    - 5.4.7 Johnson Counters . . . . . 127
    - 5.4.8 Parameterized Counter. . . . . 130
  - 5.5 Shift Register. . . . . 130
    - 5.5.1 Right and Left Shift . . . . . 130
    - 5.5.2 Parallel Input and Parallel Output (PIPO) Shift Register. . . . . 132
  - 5.6 Timing and Performance Evaluation . . . . . 138
  - 5.7 Asynchronous Counter Design. . . . . 138
    - 5.7.1 Ripple Counters . . . . . 140
  - 5.8 Memory Modules and Design . . . . . 140
  - 5.9 Summary . . . . . 144
- 6 Sequential Design Guidelines . . . . . 145**
  - 6.1 Use of Blocking Assignments . . . . . 146
    - 6.1.1 Blocking Assignments and Multiple “Always” Blocks. . . . . 146
    - 6.1.2 Blocking Assignments in the Same “Always” Block . . . . . 146
    - 6.1.3 Example Blocking Assignment. . . . . 149

6.2	Nonblocking Assignments . . . . .	150
6.2.1	Example Nonblocking Assignment . . . . .	150
6.2.2	Ordering on Non-blocking Assignments . . . . .	153
6.3	Latch Versus Flip-Flop . . . . .	154
6.3.1	D Flip-Flop . . . . .	154
6.3.2	Latch . . . . .	154
6.4	Use of Synchronous Versus Asynchronous Reset . . . . .	156
6.4.1	Asynchronous Reset D Flip-Flop . . . . .	157
6.4.2	Synchronous Reset D Flip_Flop . . . . .	157
6.5	Use of If-Else Versus Case Statements . . . . .	157
6.6	Internally Generated Clocks . . . . .	157
6.7	Gated Clocks . . . . .	161
6.8	Use of Pipelining in Design . . . . .	161
6.8.1	Design Without Pipelining . . . . .	162
6.8.2	Design with Pipelining . . . . .	163
6.9	Guidelines for Modeling Synchronous Designs . . . . .	163
6.10	Multiple Clocks in the Same Module . . . . .	163
6.11	Multi Phase Clocks in the Design . . . . .	165
6.12	Guidelines for Modeling Asynchronous Designs . . . . .	169
6.13	Summary . . . . .	169
<b>7</b>	<b>Complex Designs Using Verilog RTL . . . . .</b>	<b>171</b>
7.1	ALU Design . . . . .	172
7.1.1	Logical Unit Design . . . . .	172
7.1.2	Arithmetic Unit . . . . .	179
7.1.3	Arithmetic and Logical Unit . . . . .	181
7.2	Functions and Tasks . . . . .	184
7.2.1	Counting 1's from the Given String . . . . .	185
7.2.2	Module to Count 1's using Functions . . . . .	185
7.3	Parity Generators and Detectors . . . . .	187
7.3.1	Parity Generator . . . . .	187
7.3.2	Add_Sub_Parity Checker . . . . .	189
7.4	Barrel Shifters . . . . .	192
7.5	Summary . . . . .	195
<b>8</b>	<b>Finite State Machines . . . . .</b>	<b>197</b>
8.1	Moore Versus Mealy Machines . . . . .	198
8.1.1	Level to Pulse Converter . . . . .	200
8.2	FSM Encoding Styles . . . . .	205
8.2.1	Binary Encoding . . . . .	206
8.2.2	Gray Encoding . . . . .	208
8.3	One-Hot Encoding . . . . .	210
8.4	Sequence Detectors Using FSMs . . . . .	212

- 8.4.1 Sequence Detector Using Mealy Machine Two  
Always Blocks . . . . . 212
- 8.4.2 Sequence Detector Using Mealy Machine  
for ‘101’ Sequence . . . . . 215
- 8.5 Improving the Design Performance for FSMs . . . . . 215
- 8.6 Summary . . . . . 217
- 9 Simulation Concepts and PLD-Based Designs . . . . . 219**
- 9.1 Key Simulation Concepts . . . . . 219
  - 9.1.1 Simulation for Blocking and Nonblocking  
Assignments . . . . . 220
  - 9.1.2 Blocking Assignments with Inter-assignment  
Delays. . . . . 222
  - 9.1.3 Blocking Assignments with Intra-assignment  
Delays. . . . . 224
  - 9.1.4 Nonblocking Assignments with Inter-assignment  
Delays. . . . . 224
  - 9.1.5 Nonblocking Assignments with Intra-assignment  
Delays. . . . . 226
- 9.2 Simulation Using Verilog . . . . . 226
- 9.3 Introduction to PLD . . . . . 230
- 9.4 FPGA as Programmable ASIC. . . . . 233
  - 9.4.1 SRAM Based FPGA. . . . . 233
  - 9.4.2 Flash-Based FPGA . . . . . 233
  - 9.4.3 Antifuse FPGAS. . . . . 234
  - 9.4.4 FPGA Building Blocks . . . . . 235
- 9.5 FPGA Design Flow . . . . . 236
  - 9.5.1 Design Entry . . . . . 237
  - 9.5.2 Design Simulation and Synthesis . . . . . 238
  - 9.5.3 Design Implementation . . . . . 238
  - 9.5.4 Device Programming. . . . . 238
- 9.6 Logic Realization Using FPGA . . . . . 239
  - 9.6.1 Configurable Logic Block . . . . . 239
  - 9.6.2 Input–Output Block (IOB). . . . . 240
  - 9.6.3 Block RAM. . . . . 241
  - 9.6.4 Digital Clock Manager (DCM) Block . . . . . 242
  - 9.6.5 Multiplier Block. . . . . 242
- 9.7 Design Guidelines for FPGA-Based Designs . . . . . 243
  - 9.7.1 Verilog Coding Guidelines. . . . . 243
  - 9.7.2 FSM Guidelines . . . . . 244
  - 9.7.3 Combinational Design and Combinational Loops . . . . . 245
  - 9.7.4 Grouping the Terms . . . . . 245
  - 9.7.5 Assignments . . . . . 245
  - 9.7.6 Simulation and Synthesis Mismatch . . . . . 245
  - 9.7.7 Post-synthesis Verification . . . . . 246

- 9.7.8 Guidelines for Area Optimization . . . . . 246
- 9.7.9 Guidelines for Clock . . . . . 247
- 9.7.10 Synchronous Versus Asynchronous Designs. . . . . 248
- 9.7.11 Guidelines for Use of Reset . . . . . 249
- 9.7.12 Guidelines for CDC . . . . . 250
- 9.7.13 Guidelines for Low Power Design . . . . . 251
- 9.7.14 Guidelines for Use of Vendor-Specific IP Blocks . . . 251
- 9.8 Summary . . . . . 252
- References . . . . . 253
- 10 ASIC RTL Synthesis . . . . . 255**
- 10.1 What Is ASIC? . . . . . 256
  - 10.1.1 Full-Custom ASIC . . . . . 256
  - 10.1.2 Standard Cell ASIC . . . . . 256
  - 10.1.3 Gate Array ASIC . . . . . 257
- 10.2 ASIC Design Flow . . . . . 257
  - 10.2.1 Design Specification . . . . . 257
  - 10.2.2 RTL Design and Verification . . . . . 259
  - 10.2.3 ASIC Synthesis . . . . . 259
  - 10.2.4 Physical Design and Implementation . . . . . 260
- 10.3 ASIC Synthesis Using Design Compiler . . . . . 261
- 10.4 ASIC Synthesis Guidelines . . . . . 263
- 10.5 Constraining Design Using Synopsys DC . . . . . 264
  - 10.5.1 Reading the Design. . . . . 264
  - 10.5.2 Checking of the Design. . . . . 265
  - 10.5.3 Clock Definitions . . . . . 265
  - 10.5.4 Skew Definition . . . . . 266
  - 10.5.5 Defining Input and Output Delay . . . . . 267
  - 10.5.6 Defining Minimum (Min) and Maximum (Max) Delay . . . . . 267
  - 10.5.7 Design Synthesis . . . . . 267
  - 10.5.8 Saving the Design. . . . . 267
- 10.6 Synthesis Optimization Techniques. . . . . 268
  - 10.6.1 Resource Allocation . . . . . 269
  - 10.6.2 Common Factors and Sub-expressions Use for Optimization. . . . . 270
  - 10.6.3 Moving the Piece of Code . . . . . 271
  - 10.6.4 Constant Folding . . . . . 272
  - 10.6.5 Dead Zone Elimination . . . . . 273
  - 10.6.6 Use of Parentheses . . . . . 273
  - 10.6.7 Partitioning and Structuring the Design . . . . . 274
- 10.7 Summary . . . . . 274
- Reference . . . . . 275

- 11 Static Timing Analysis** . . . . . 277
  - 11.1 Setup Time . . . . . 278
  - 11.2 Hold Time. . . . . 279
  - 11.3 Clock to Q Delay. . . . . 280
    - 11.3.1 Frequency Calculations . . . . . 280
  - 11.4 Skew in Design . . . . . 282
  - 11.5 Timing Paths in Design. . . . . 284
    - 11.5.1 Input-to-Register Path . . . . . 284
    - 11.5.2 Register-to-Output Path . . . . . 284
    - 11.5.3 Register-to-Register Path . . . . . 285
    - 11.5.4 Input-to-Output Path . . . . . 286
  - 11.6 Timing Goals for the Design . . . . . 286
  - 11.7 Min-Max Analysis for ASIC Design. . . . . 287
  - 11.8 Fixing Design Violations. . . . . 289
    - 11.8.1 Changes at the Architecture Level. . . . . 289
    - 11.8.2 Changes at Microarchitecture Level . . . . . 290
    - 11.8.3 Optimization During Synthesis . . . . . 290
  - 11.9 Fixing Setup Violations in the Design. . . . . 291
    - 11.9.1 Logic Duplication. . . . . 291
    - 11.9.2 Encoding Methods . . . . . 292
    - 11.9.3 Late Arrival Signals . . . . . 293
    - 11.9.4 Register Balancing . . . . . 293
  - 11.10 Hold Violation Fix . . . . . 294
  - 11.11 Timing Exceptions in the Design . . . . . 295
    - 11.11.1 Asynchronous and False Paths . . . . . 295
    - 11.11.2 Multicycle Paths. . . . . 296
  - 11.12 Pipelining and Performance Improvement . . . . . 297
  - 11.13 Summary . . . . . 298
  - Reference . . . . . 298
  
- 12 Constraining ASIC Design** . . . . . 299
  - 12.1 Introduction to Design Constraints . . . . . 300
  - 12.2 Compilation Strategy . . . . . 304
    - 12.2.1 Top-Down Compilation. . . . . 304
    - 12.2.2 Bottom-Up Compilation . . . . . 305
  - 12.3 Area Minimization Techniques. . . . . 306
    - 12.3.1 Avoid Use of Combinational Logic  
as Individual Block. . . . . 306
    - 12.3.2 Avoid Use of Glue Logic Between  
Two Modules. . . . . 307
    - 12.3.3 Use of set\_max\_area Attribute . . . . . 308
    - 12.3.4 Area Report. . . . . 308
  - 12.4 Timing Optimization and Performance Improvement. . . . . 309
    - 12.4.1 Design Compilation with ‘map\_effort high’ . . . . . 309
    - 12.4.2 Logical Flattening. . . . . 309

12.4.3	Use of group_path Command . . . . .	310
12.4.4	Submodule Characterizing . . . . .	311
12.4.5	Register Balancing . . . . .	313
12.4.6	FSM Optimization . . . . .	314
12.4.7	Fixing Hold Violations . . . . .	315
12.4.8	Report Command . . . . .	315
12.5	Constraint Validation . . . . .	318
12.6	Commands for the DRC, Power, and Optimization . . . . .	318
12.7	Summary . . . . .	319
	References . . . . .	320
<b>13</b>	<b>Multiple Clock Domain Design . . . . .</b>	<b>321</b>
13.1	What Is Multiple Clock Domain? . . . . .	322
13.2	What Is Clock Domain Crossing (CDC) . . . . .	322
13.3	Level Synchronizers . . . . .	327
13.4	Pulse Synchronizers . . . . .	330
13.5	MUX Synchronizer . . . . .	331
13.6	Challenges in the Design of Synchronizers . . . . .	331
13.7	Data Path Synchronizers . . . . .	338
	13.7.1 Handshaking Mechanism . . . . .	338
	13.7.2 FIFO Synchronizer . . . . .	340
	13.7.3 Gray Encoding . . . . .	341
13.8	Design Guidelines for the Multiple Clock Domain Designs . . . . .	343
13.9	FIFO Depth Calculations . . . . .	347
13.10	Case Study . . . . .	351
13.11	Summary . . . . .	358
<b>14</b>	<b>Low Power Design . . . . .</b>	<b>359</b>
14.1	Introduction to Low Power Design . . . . .	359
14.2	Power Dissipation in CMOS Inverter . . . . .	360
14.3	Switching and Leakage Power Reduction Techniques . . . . .	363
	14.3.1 Clock Gating and Clock Tree Optimizations . . . . .	364
	14.3.2 Operand Isolations . . . . .	364
	14.3.3 Multiple $V_{th}$ . . . . .	364
	14.3.4 Multiple Supply Voltages (MSV) . . . . .	364
	14.3.5 Dynamic Voltage and Frequency Scaling (DVFS) . . . . .	365
	14.3.6 Power Gating (Power Shut-Off) . . . . .	365
	14.3.7 Isolation Logic . . . . .	365
	14.3.8 State Retention . . . . .	366
14.4	Low Power Design Techniques at the RTL Level . . . . .	366

- 14.5 Low Power Design Architecture and UPF Case Study . . . . . 370
  - 14.5.1 Isolation Cells . . . . . 371
  - 14.5.2 Retention Cells. . . . . 372
  - 14.5.3 Level Shifters. . . . . 374
  - 14.5.4 Power Sequencing and Scheduling . . . . . 374
- 14.6 Summary . . . . . 380
- References . . . . . 380
- 15 System on Chip (SOC) Design . . . . . 381**
  - 15.1 What is System on Chip (SOC)? . . . . . 382
  - 15.2 SOC Architecture. . . . . 382
  - 15.3 SOC Design Flow . . . . . 383
    - 15.3.1 IP Design and Reuse. . . . . 383
    - 15.3.2 SOC Design Considerations . . . . . 385
    - 15.3.3 Hardware Software Codesign . . . . . 386
    - 15.3.4 Interface Timings . . . . . 386
    - 15.3.5 EDA Tool and License Requirements . . . . . 387
    - 15.3.6 Developing the Required Prototyping Platform. . . . . 387
    - 15.3.7 Developing the Test Plan. . . . . 388
    - 15.3.8 Developing the Verification Environment. . . . . 388
    - 15.3.9 Prototyping Using FPGAs . . . . . 388
    - 15.3.10 ASIC Porting . . . . . 389
  - 15.4 SOC Design Challenges . . . . . 390
  - 15.5 Case Study . . . . . 392
  - 15.6 SOC Design Blocks . . . . . 392
    - 15.6.1 Microprocessors or Microcontrollers . . . . . 392
    - 15.6.2 Counters and Timers. . . . . 393
    - 15.6.3 General Purpose IO Block . . . . . 395
    - 15.6.4 Universal Asynchronous Receiver and Transmitter (UART). . . . . 396
    - 15.6.5 Bus Arbitration Logic . . . . . 397
  - 15.7 Summary . . . . . 398
- Appendix I: Synthesizable and Non-Synthesizable Verilog Constructs . . . . . 399**
- Appendix II: Xilinx Spartan Devices . . . . . 401**
- Appendix III: Design For Testability . . . . . 405**
- Index . . . . . 409**

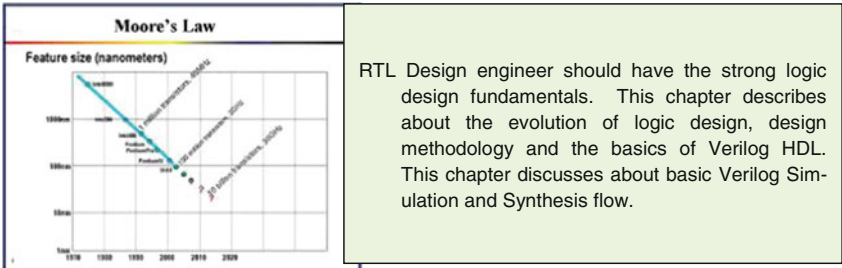


## About the Author

**Vaibbhav Taraate** is Entrepreneur and Mentor at “Semiconductor Training @ Rs.1”. He obtained a BE (Electronics) degree from Shivaji University, Kolhapur in 1995 and secured a gold medal for standing first in all engineering branches. He has completed his MTech (Aerospace Control and Guidance) in 1999 from IIT Bombay. He has over 15 Years of experience in semi-custom ASIC and FPGA design, primarily using HDL languages such as Verilog and VHDL. He has worked with few multinational corporations as consultant, senior design engineer, and technical manager. His areas of expertise include RTL design using VHDL, RTL design using Verilog, complex FPGA-based design, low power design, synthesis/optimization, static timing analysis, system design using microprocessors, high speed VLSI designs, and architecture design of complex SOCs.

# Chapter 1

## Introduction



**Abstract** This chapter discusses about the overview of the design abstraction levels and the evolution of logic design in the perspective of the system design. This chapter is mainly focused on the familiarity with Verilog HDL, different modeling styles, and Verilog operators. The chapter is organized in such a way that it covers basic to the practical scenarios in detail. All the Verilog operators with meaningful examples are described in this chapter for easy understanding.

**Keywords** RTL · IEEE 1364-2005 · Behavioral model · Structural model · Verilog · VHDL · Moore's law · Concurrent · Sequential · Procedural blocks · Always · Four-value logic · Operators · Arithmetic · Shift · Logical · Bitwise · Concatenation · Case equality · Case inequality · Continuous assignments · Net · Variable · Data types

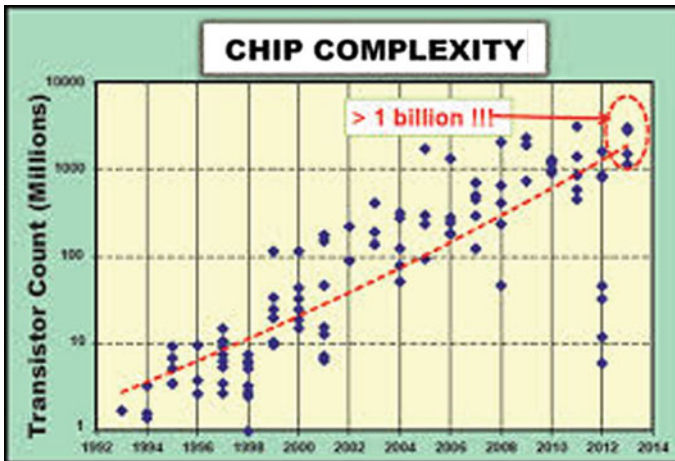
### 1.1 Evolution of Logic Design

During the year 1958, Jack Kilby, a young electrical engineer at Texas Instrument figured out how to place the circuit elements, transistors, resistors, and capacitors, on small piece of Germanium. But prior to the year 1958, many more revolutionized ideas were published and conceptualized.

Gottfried Leibniz was a famous mathematician and philosopher from Germany and he redefined the binary number system during the year 1676–1679. After the successful redefinition of number systems, the famous mathematician George Boole during year 1854 invented the Boolean algebra and the revolution of the digital logic design set into motion.

The actual invention of the prototype transistor model during the year 1946–1947 at Bell Labs by Shockley, Bardeen, Brattain had revolutionized the use of semiconductor in switching theory and for design of chip. The design of first working transistor was the biggest contribution by the Morris Tanenbaum during the year 1954 at Texas Instruments.

The invention of CMOS logic during 1963 has made integration of logic cells very easy and it was predicted by Intel's cofounder Gordon Moore that the density of the logic cells for the same silicon area will be doubled for every 18 to 24 months. This is what we call as Moore's law.



How Moore's prediction was right, that experience engineers can get with the complex VLSI-based ASIC chip designs. In the present decade, the chip area has shrunk enough and process technology node on which design houses foundries are working is 14 nm and the chip has billions of cells of small silicon die size. With the evolutions in the design and manufacturing technologies; most of the designs are implemented by using Very High Speed Integrated Circuit Hardware Description Language (*V<sub>H</sub>SIC<sub>H</sub>DL*) or using Verilog. We are focusing on the Verilog as hardware description language. The evolution in the EDA industry has opened up new efficient path ways for the design engineers to complete the milestones in less time.

## 1.2 System and Logic Design Abstractions

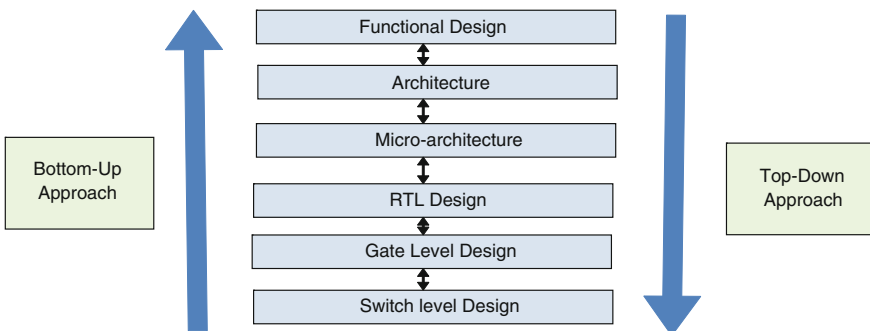
As shown in Fig. 1.1 most of the designs have various abstraction levels. The design approach can be top-down or bottom-up. The implementation team takes decision about the right approach depending on the design complexity and availability of design resources. Most of the complex designs are using the top-down approach instead of bottom-up approach.

The design is described as functional model initially and the architecture and micro-architecture of the design is described by understanding the functional design specifications. Architecture design involves the estimation of the memory processor logic and throughput with associative glue logic and functional design requirements. Architecture design is in the form of functional blocks and represents the functionality of design in the block diagram form.

The micro-architecture is the detailed representation of every architecture block and it describes the block and sub block level details, interface and pin connections, and hierarchical design details. The information about synchronous or asynchronous designs and clock and reset trees can be also described in the micro-architecture document.

RTL stands for Register Transfer Level. RTL design uses micro-architecture as reference design document and design can be coded using Verilog RTL for the required design functionality. The efficient design and coding guidelines at this stage plays important role and efficient RTL reduces the overall time requirement during the implementation phase. The outcome of RTL design is gate level netlist. Gate level netlist is the output from the RTL design stage after performing RTL synthesis and it is representation of the functional design in the form of combinational and sequential logic cells.

Finally, the switch level design is the abstraction used at the layout to represent the design in the form of switches. PMOS, NMOS, and CMOS.



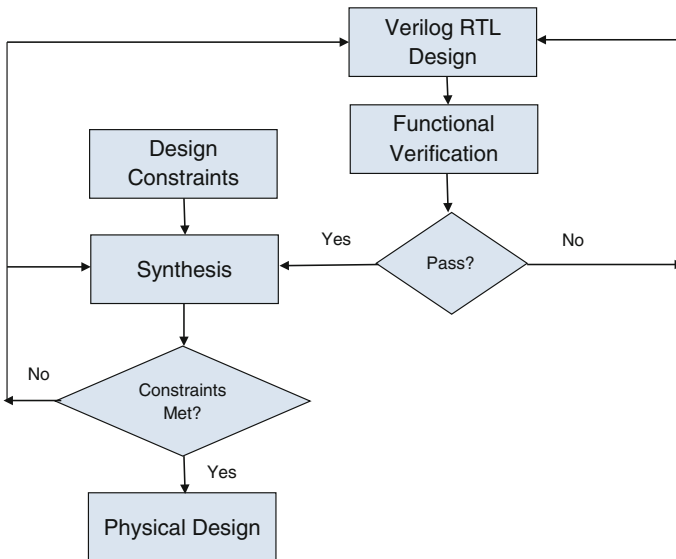
**Fig. 1.1** Design abstractions

### 1.3 Integrated Circuit Design and Methodologies

With the evolution of VLSI design technology, the designs are becoming more complex and SOC-based designs are feasible in shorter design cycle time. The demand of the customers to get products in the shorter design cycle time is possible by using efficient design flow. The design needs to be evolved from specification stage to final layout. The use of EDA tools with the suitable features has made it possible to have the bug free designs with proven functionality. The design flow is shown in Fig. 1.2 and it consist of three major steps to generate the netlist.

#### 1.3.1 RTL Design

Functional design is described in the document form using the architecture and micro-architecture. The RTL design using Verilog uses the micro-architecture document to code the design. RTL designer uses the suitable design and coding guidelines while implementing the RTL design. An efficient RTL design always plays important role during implementation cycle. During this, the designer describes the block level and top level functionality using an efficient Verilog RTL.



**Fig. 1.2** Simulation and synthesis flow

### ***1.3.2 Functional Verification***

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

### ***1.3.3 Synthesis***

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

### ***1.3.4 Physical Design***

It involves the floor-planning of design, power planning, place and route, Clock tree synthesis, post layout verification, Static timing analysis, and generation of GDSII for an ASIC design. This step is out of scope for the subsequent discussions.

## **1.4 Verilog HDL**

Verilog is standardized as IEEE 1364 standard and used to describe digital electronic circuits. Verilog HDL is used mainly in design and verification at the RTL level of abstraction. Verilog was created by Prabhu Goel and Phil Moorby during

the year 1984 at Gateway design automations. Verilog IEEE standards are Verilog-95 (IEEE 1364-1995), Verilog-2001 (IEEE 1364-2001), and Verilog-2005 (IEEE 1364-2005). Verilog is case sensitive and before we proceed further to discuss on RTL design and synthesis, it is essential to have the basic understanding of the Verilog code structure (Fig. 1.3).

As shown in the Verilog code structure template.

```
// indicates the comment line
```

*<module\_name>* is the name of module. Give some meaningful name while declaring module.

*<port\_name>* is the name of input or output port.

*<size>* is the width of the input port , output port or net

*wire* and *reg* are net types, *wire* doesn't hold any data and used in continuous assignment. *Reg* is used to hold data and used for the procedural assignments.

*<net\_name>* is the name of net declared

*always* and *assign* are keyword and used to describe the design functionality.

*assign statements are continuous assignments and executes in parallel.*

*always block is procedural block and all the statements inside always block are executed sequentially. Multiple always blocks are executed concurrently.*

*endmodule* key word and indicates the end of the design module!

Every Verilog code starts with the 'module' keyword and ends with "end-module." Module consists of the port declaration, net declaration, and the functionality of design.

```

// Verilog code structure
// Verilog is case sensitive language
// Verilog code starts with definition of module
module <module_name> ( <Input, output port List>); // module is keyword

input <size> <port_name>;
input <port_name>;
output <size> <port_name>;
output <port_name>;
wire <size> <net_name>;
wire net_name;
reg <size> <net_name or port name>;
reg <net_name or port_name>;

// Functionality of design
always@( <list of signals, nets, ports>) // procedural block for design functionality
begin
    // write code here
end

assign <port_name or net_name> = // write the functional expression;
endmodule
    
```

Verilog code starts with the keyword module. Module declaration consists of the module name and port list.

Define the inputs, outputs and internal nets using keywords input, output, wire or reg respectively. Wire is used for the continuous assignment and reg is used in the procedural blocks. inout can be used for the bidirectional port.

Design functionality can be written by using single or multiple procedural blocks or by using the continuous assignment statements.

Fig. 1.3 Verilog code structure template

## 1.5 Verilog Design Description

In the practical scenarios the Verilog HDL is categorized into three different kind of coding descriptions. The different styles of coding description are structural, behavioral, and synthesizable RTL. Consider the design structure of half adder shown in Fig. 1.4c which describes different coding styles. Figure 1.4 shows the truth table, schematic and logic structure realization for half adder.

### 1.5.1 Structural Design

Structural design defines a data structure of the design and it is described in the form of netlist using the required net connectivity. Structural design is mainly the



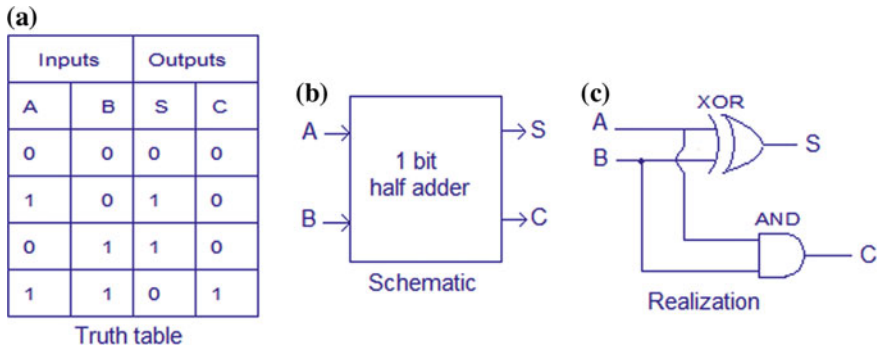


Fig. 1.4 Logic structure for “basic\_Verilog”

instantiation of different small complexity digital logic blocks. It is basically a design connection of small modules to realize moderate or complex logic. Example 1.1 describes the structural code style for “basic\_verilog” module (Fig. 1.4).

```
// Verilog structural code style
module basic_verilog (A,B,S,C) ;
input A;
input B;
output S;
output C;
wire A, B;
wire S, C;

// Functionality of design
xor_gate U1 (.A(A), .B(B), .S(S));
and_gate U2 (.A(A), .B(B), .C(C));

endmodule
```

← Declare Verilog module 'basic\_verilog' with input ports 'A', 'B' and output ports 'S', 'C'

← Instantiation of the components xor\_gate and and\_gate". It is assumed that the precompiled xor\_gate and and\_gate is available in the library.

Example 1.1 Structural style for “basic\_verilog”

```

// Verilog behavior code style
module basic_verilog (A,B,S,C);
input A, B;
output S, C;
reg S, C;
// Functionality of design
always@(A or B)
if (A==B)
S= 1'b0;
else
S=1'b1;
always@(A or B)
if (A & B)
C=1'b1;
else
C=1'b0;
endmodule

```

Declare verilog module 'basic\_verilog' with input ports 'A', 'B' and output ports 'S', 'C'

Sensitive to 'a' or 'b' and described in sensitivity list.

The description using 'if-else' statement describes the output assignment to 'S'. When both inputs 'A', 'B' are at same logic level output assigned to 'S' is logical '0' else output assignment to 'S' is logical '1'

The description using 'if-else' statement describes the output assignments to 'C'. When both 'A', 'B' are at logic '1' level then output assigned to 'C' is logic '1' else output assigned to 'C' is logic '0'.

**Example 1.2** Behavior style for basic\_verilog

### 1.5.2 Behavior Design

The name itself indicates the nature of coding style. In the behavior style of Verilog code, the functionality is coded from the truth table for the specific design. It is assumed that the design is black box with the inputs and outputs. The main intention of the designer is to map the functionality at output according to the required set of inputs (Example 1.2).

```

// Verilog synthesizable RTL code style
module basic_verilog (A,B,S,C) ;

input A;
input B;
output S;
output C;

reg S;
reg C;

// Functionality of design
always s@ (A or B)
begin
S= A ^ B;
C= A & B;
end
endmodule

```

Declare Verilog module 'basic\_verilog' with input ports 'A', 'B' and output ports 'S', 'C'.

Description of outputs 'S', 'C' in the form of logical expressions and used for less complex designs!

**Example 1.3** Synthesizable RTL Verilog code for “basic\_verilog”

### 1.5.3 Synthesizable RTL Design

Synthesizable RTL code is used in the practical environment to describe the functionality of design using synthesizable constructs. The RTL code style is high-level description of functionality using synthesizable constructs. The RTL coding style is recommended using the synthesizable Verilog constructs (Example 1.3).

## 1.6 Key Verilog Terminologies

Before the subsequent discussion on Verilog terminologies, it is essential to understand how Verilog works. Why it is a hardware description language?

- Verilog is different from the software languages as it is used to describe the hardware. Verilog supports describing the propagation time and sensitivity.
- Verilog supports concurrent (parallel) execution of statements and even sequential execution of statements.

- Verilog supports blocking ('=') assignments and even nonblocking assignments ('<='). Blocking assignments are used to describe combinational logic and nonblocking assignments are used to describe sequential logic. These assignments will be discussed in subsequent chapters.
- Verilog supports the declaration of input, output, and bidirectional (inout) ports.
- Verilog supports definition of constants and parameters. Verilog supports file handling.
- Verilog supports four-value logic: logical '0', logical '1', high impedance 'z', and unknown 'X'.
- Verilog supports procedural blocks using "always" and "initial" keywords. Procedural block with keyword "always" indicates free running process and executes always on event and procedural block with "initial" keyword indicates the execution of block only once. Both procedural blocks executes at simulator time '0'. These blocks will be discussed in the subsequent chapters.
- Verilog supports synthesizable constructs as well as non-synthesizable constructs.
- Verilog supports use of tasks and functions for recursive use.
- Verilog supports Program Language Interface (PLI) to transfer control from Verilog to functions written in 'C' language.

The template shown below describes key Verilog constructs used to describe most of the combinational logic design (Fig. 1.5).

### ***1.6.1 Verilog Arithmetic Operators***

Verilog supports addition, subtraction, multiplication and division, and modulus operators to perform arithmetic operations. Table 1.1 describes the arithmetic operators (Example 1.4).

### ***1.6.2 Verilog Logical Operators***

Verilog supports logical AND, OR, and negation operators to perform desired logical operation. Logical operators are used to return single bit value at the end of the operation. Table 1.2 describes the functional use of logical operators (Example 1.5).

### ***1.6.3 Verilog Equality and Inequality Operators***

Verilog equality operators are used to return true or false value after comparing two operands. Table 1.3 describes the functionality of the operators (Example 1.6).

1. Continues assignment using 'assign'

```
// consider 'y' is declared as output port and 'a', 'b' as input port
assign y = a;           // assigns input 'a' to output 'y'
assign y = a ^ b;      // assigns XOR of 'a', 'b' to output 'y'
assign y = ~a;         // assigns NOT of 'a' to output 'y'
```

2. Use of 'always' procedural block

```
// Consider input ports are declared as 'a', 'b', 's' and output port is declared as 'y'
always (a or b or s)
if (s)
y = a;
else
y = b;
```

Procedural block 'always' executes when there is an event of either 'a', 'b' or 's'. If 's' is equal to logical '1' then 'a' is assigned to output 'y'.  
If 's' is equal to logical '0' then 'b' is assigned to output 'y'.  
If-else is used for assigning the output value depending on the true or false condition.

3. Declaration using 'wire'

```
// 'wire' and 'reg' are used to declare the nets and reg variable respectively.
wire y = a ^ b;
```

4. Declaration using 'reg'

```
// Consider 'y' is declared as an output port and used in 'always' block
reg y; // used as variable and to be declared in 'always' block
always @ ( a or b or s)
y = (s) ? a : b;
```

5. Use of 'case' construct

```
// Consider input ports are declared as 'a', 'b', 's' and output port is declared as 'y'
always (a or b or s)
case (s)
1'b0 : y = a;
1'b1 : y = b;
endcase
```

Procedural block 'always' executes when there is an event of either 'a', 'b' or 's'. If 's' is equal to logical '1' then 'b' is assigned to output 'y'. 'case' is used as control flow construct to assign output 'y' depending on 's'. If 's' is equal to logical '0' then 'a' is assigned to output 'y'. If-else is used for assigning the output value depending on the true or false condition.

Fig. 1.5 Basic Verilog definitions and descriptions

**Table 1.1** Verilog arithmetic operators

Operator	Name	Functionality
+	Binary addition	To perform addition of two binary operands
-	Binary minus	To perform subtraction of two binary operands
*	Multiplication	To perform multiplication of two binary operands
/	Division	To perform division of two binary operands
%	Modulus	To find modulus from division of two operands

```

//Verilog Arithmetic Operator

module arithmetic_operation (a, b, y1, y2, y3, y4, y5);

input [3:0] a, b;

output [4:0] y1;

output [5:0] y3;

output [3:0] y2, y4, y5;

reg [4:0] y1;

reg [5:0] y3;

reg [3:0] y2, y4, y5;

always @ (a or b)

begin

    y1 = a + b;

    y2 = a - b;

    y3 = a*b;

end

endmodule
    
```

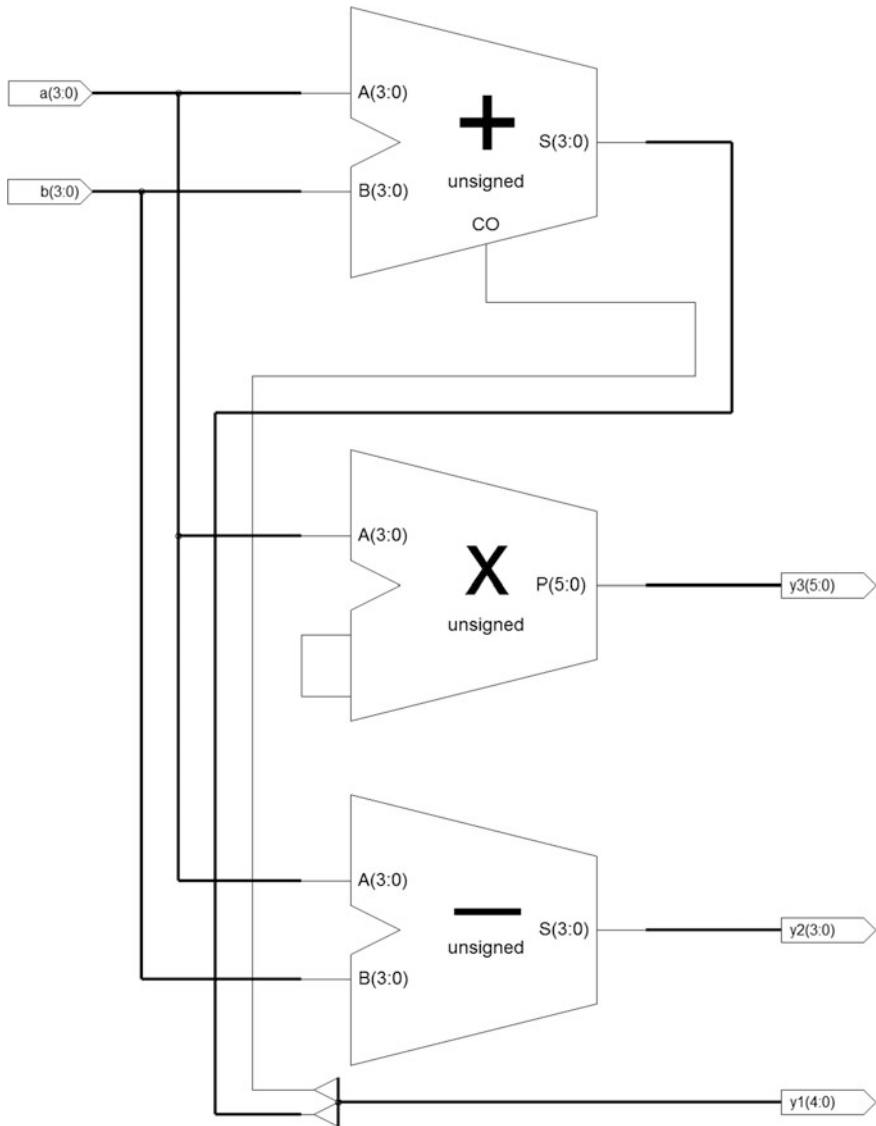


After the arithmetic operations the respective output 'y' is assigned with the value computed.

**Example 1.4** Verilog arithmetic operators

### 1.6.4 Verilog Sign Operators

Verilog supports the operators '+' or '-' to assign sign to the operand. Table 1.4 describes the sign operands (Example 1.7).



**Example 1.4** (continued)

**Table 1.2** Verilog logical operators

Operator	Name	Functionality
&&	Logical AND	To perform logical AND on two binary operands
	Logical OR	To perform logical OR on two binary operands
!	Logical Negation	To perform logical negation for the given binary number

```

//verilog logical operator

module logical_operator (a, b, c, d, e, f, y);

input [2:0] a, b, c, d, e, f;

output y;

reg y;

always @(a or b or c or d or e or f)

begin

    if ((a < b) && ((c == d) || !(e > f)))

        y=1;

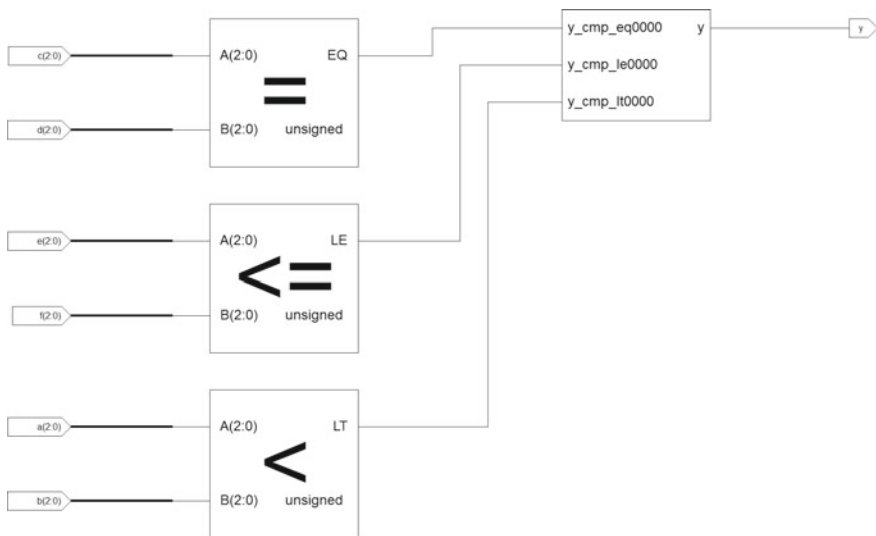
    else

        y=0;

end

endmodule
    
```

Single bit output 'y' is assigned after performing the logical operation specified in expression.



Example 1.5 Verilog logical operators

Table 1.3 Verilog equality and inequality Operators

Operator	Name	Functionality
==	Case equality	To compare the two operands
!=	Case inequality	Used to find out inequality for the two operands



**//Verilog equality Operator**

```
module equality_operator (a, b, y1, y2, y3);
```

```
input [7:0] a, b;
```

```
output y1, y2;
```

```
output [7:0] y3;
```

```
reg y1, y2;
```

```
reg [7:0] y3;
```

```
always @ (a or b)
```

```
begin
```

```
    y1 = a == b;
```

```
    y2 = a != b;
```

```
    if (a == b)
```

```
        y3 = a;
```

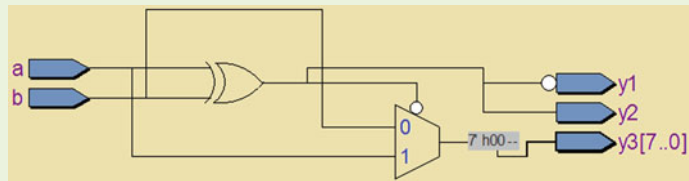
```
    else
```

```
        y3=b;
```

```
    end
```

```
endmodule
```

Comparison output true '1' or false '0' is assigned to the respective output 'y'



**Example 1.6** Verilog equality and inequality operators

### 1.6.5 Verilog Bitwise Operators

Verilog supports the bitwise operations. Logical bitwise operators use two single or multi-bit operands and return the multi-bit value. Verilog does not support NAND and NOR. Table 1.5 describes the functionality and use of bitwise operators (Example 1.8).

**Table 1.4** Verilog sign operators

Operator	Name	Functionality
+	Unary sign plus	Assign positive sign to singular operand
-	Unary sign minus	To assign negative sign to singular operand

**//Verilog sign Operator**

```

module sign_operators (a, b, y1, y2);

  input [1:0] a, b;

  output [3:0] y1, y2;

  reg [3:0] y1, y2;

  always @ (a or b)

  begin

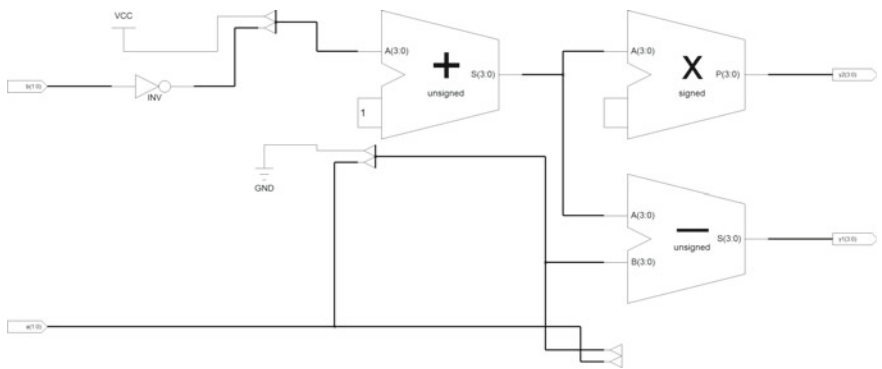
    y1 = -a+-b;

    y2 =a*-b;

  end

endmodule
    
```

After performing the specified operation on the sign numbers the output is assigned to the respective 'y1', 'y2'



**Example 1.7** Verilog sign operators

**Table 1.5** Verilog bitwise operators

Operator	Name	Functionality
&	Bitwise AND	To perform bitwise AND on two binary operands
	Bitwise OR	To perform bitwise OR on two binary operands
^	Bitwise XOR	To perform bitwise XOR on two binary operands

**//Verilog bit-wise Operator**

```

module bitwise_operator (a, b, y);

  input [6:0] a;

  input [5:0] b;

  output [6:0] y;

  reg [6:0] y;

  always @(a or b)
  begin

    y[0] = ( a[0] & b[0] );

    y[1] = ! ( a[1] & b[1] );

    y[2] = ( a[2] | b [2] );

    y[3] = ! ( a[3] & b[3] );

    y[4] = ( a[4] ^ b[4] );

    y[5] = ( a[5] ~^ b[5] );

    y[6] = !a[6] ;

  end
endmodule

```



The bit-wise result is assigned to the respective output 'y[0] to y[6]'

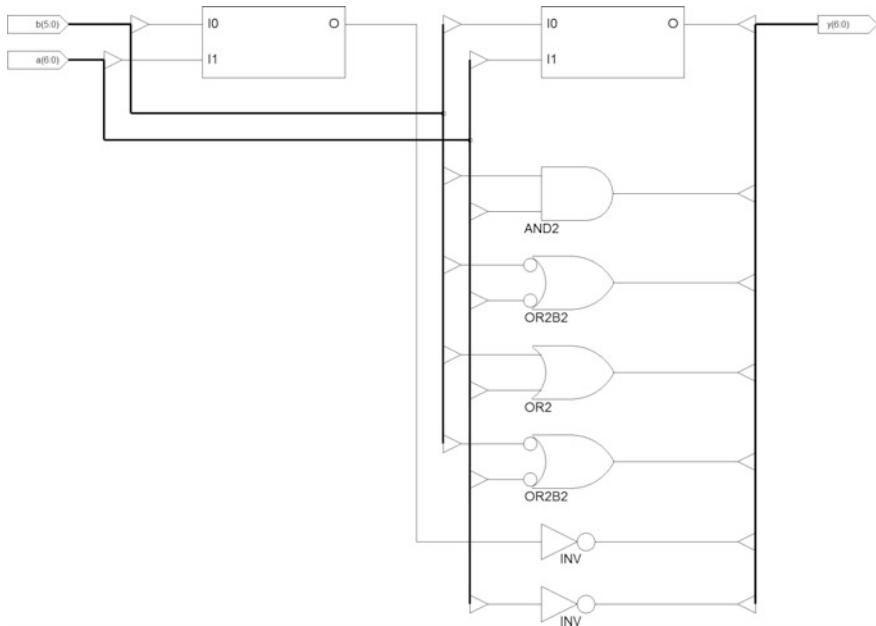
**Example 1.8** Verilog bitwise operators

### 1.6.6 Verilog Relational Operators

Verilog supports the relational operator to compare two binary numbers and returns true ('1') or false ('0') value after comparison of two operands. Table 1.6 describes the relational operators (Example 1.9).

### 1.6.7 Verilog Concatenation and Replication Operators

Verilog supports the concentration and replication for any binary string. Table 1.7 describes the functionality of concentration and replication operators (Example 1.10).



**Example 1.8** (continued)

**Table 1.6** Verilog relational operators

Operator	Name	Functionality
>	Greater than	To compare two numbers
>=	Greater than or equal to	To compare two numbers
<	Less than	To compare two numbers
<=	Less than or equal to	To compare two numbers

### 1.6.8 Verilog Reduction Operators

Verilog supports the reduction operators and returns the single bit value after bitwise reduction. Table 1.8 describes the reduction operators (Example 1.11).

```

//Verilog relational Operator

module relational_operator (a, b, y1, y2, y3, y4);

input [7:0] a, b;

output y1, y2, y3, y4;

reg y1, y2, y3, y4;

always @ (a or b)

begin

    y1 = a < b;

    y2 = a <= b;

    y4 = a > b;

    if (a > b)

        y3 =1;

    else

        y3 =0;

end

endmodule

```

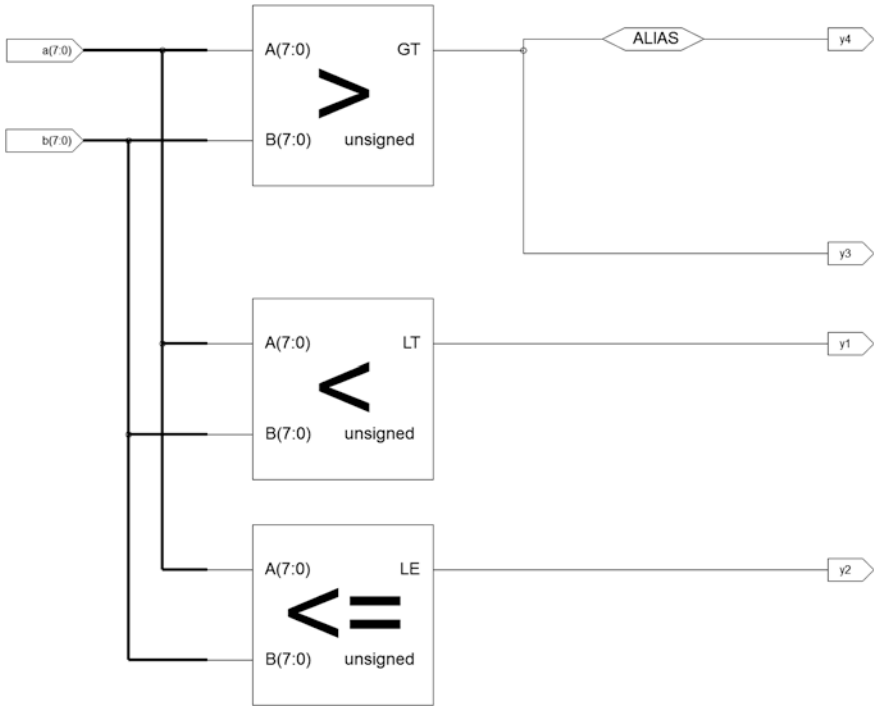


Single bit output is assigned to the respective output port after the comparison of operands, 'a', 'b'.

**Example 1.9** Verilog relational operators

### 1.6.9 Verilog Shift Operators

Verilog uses the shift operators and required two operands. These operators are used to perform the shifting operation. Table 1.9 describes the functionality of shift operators (Example 1.12).



**Example 1.9** (continued)

**Table 1.7** Verilog concentration and replication operators

Operator	Name	Functionality
{ }	Concatenation	To concatenate two binary strings
{m, {}}	Replication	To replicate the string <i>m</i> times

```

//Verilog concatenation and replication
module concatenation_operator (a, b, y);

input [2:0] a, b;

output [15:0] y

parameter c = 3'b010;

reg [15:0] y;

always @(a or b)

begin

    y = { a, b, {3{c}}, 3'b111 };

end

endmodule
    
```

← The concatenation and replication operators are used with operands 'a', 'b' and parameter value


**Example 1.10** Verilog concatenation and replication operators

**Table 1.8** Verilog reduction operators

Operator	Name	Functionality
&	Reduction AND	For performing the bitwise reduction
~&	Reduction NAND	For performing the bitwise reduction
	Reduction OR	For performing the bitwise reduction
~	Reduction NOR	For performing the bitwise reduction
^	Reduction XOR	For performing the bitwise reduction
~^ or ^~	Reduction XNOR	For performing the bitwise reduction

**//Verilog reduction Operator**

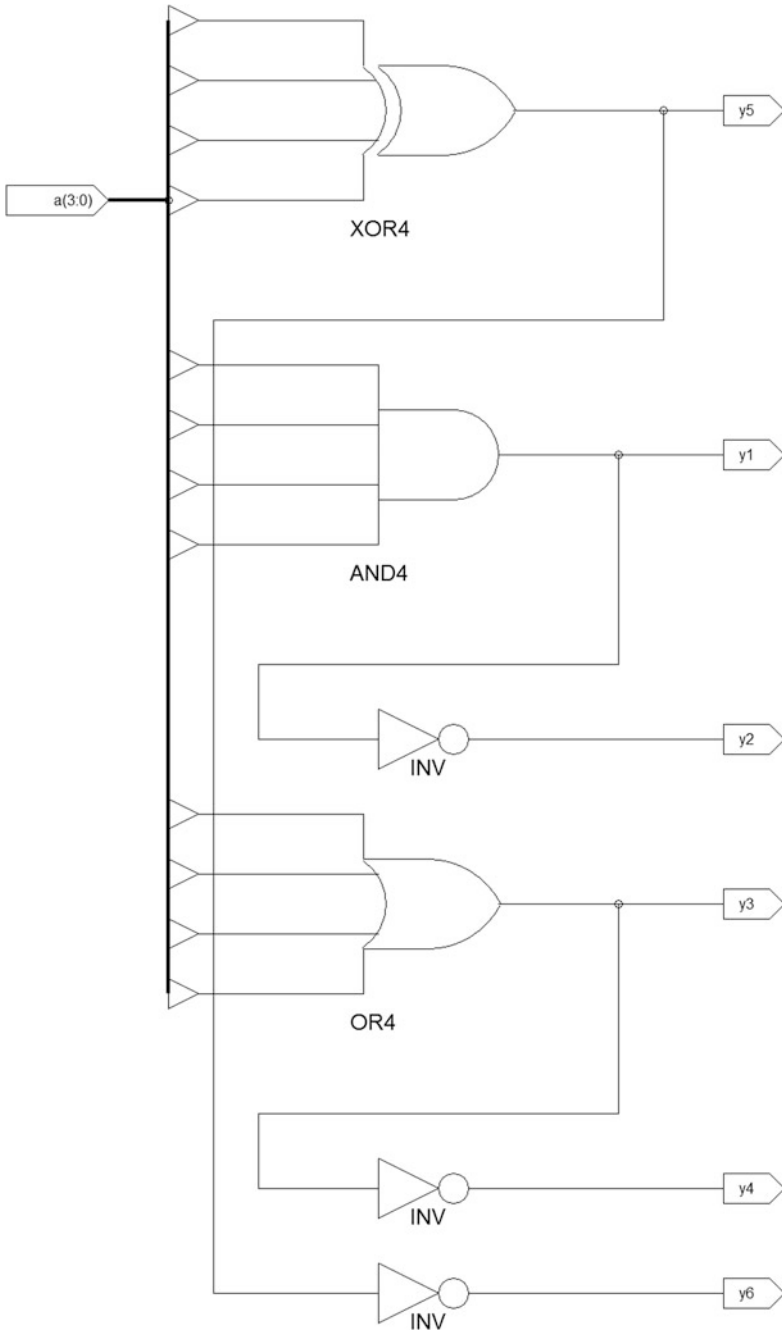
```
module reduction_operator (a, y1, y2, y3, y4, y5, y6);  
  
input [3:0] a;  
  
output y1, y2, y3, y4, y5, y6;  
  
reg y1, y2, y3, y4, y5, y6;  
  
always @ (a)  
  
begin  
  
    y1 = &a;  
  
    y2 = ~&a;  
  
    y3 = |a;  
  
    y4 = ~|a;  
  
    y5 = ^a;  
  
    y6=~^a;  
  
end  
  
endmodule
```



After performing the reduction, the single bit value is assigned to the respective output

**Example 1.11** Verilog reduction operators





Example 1.11 (continued)

**Table 1.9** Verilog shift operators

Operator	Name	Functionality
<<	Shift left	To perform logical shift left
>>	Shift right	To perform logical shift right

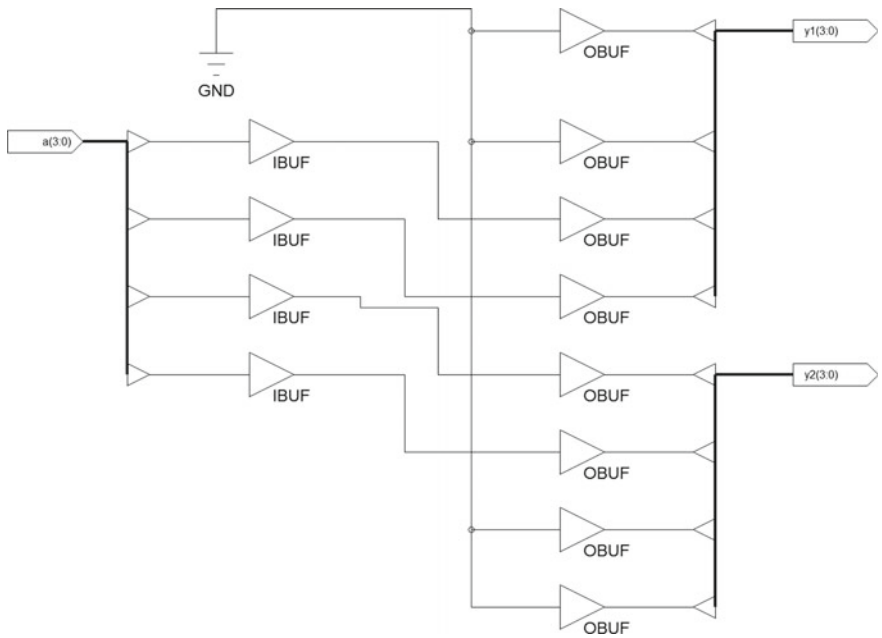
**//Verilog shift Operator**

```

module shift_operator (a, y1, y2);
input [3:0] a;
output [3:0] y1, y2;
reg [3:0] y1, y2;
parameter b=2;
always @ (a)
begin
    y1 = a << b;
    y2 = a >> b;
end
endmodule
    
```

The 'y1' is assigned to the value computed after performing left shift.

The 'y2' is assigned to the value computed after performing logical right shift.



**Example 1.12** Verilog shift operators

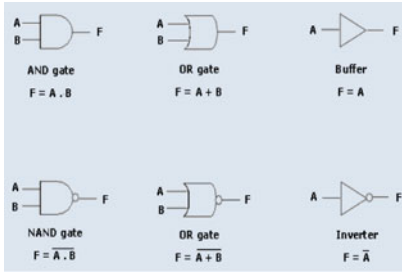
## 1.7 Summary

As discussed earlier, Verilog is a case-sensitive language and is used for design and verification of logic circuits. Following are key points to summarize this chapter.

1. Verilog is an efficient hardware description language to describe the design functionality.
2. Although there are different description styles, practically the designer uses the RTL coding style to code the RTL. Verilog supports concurrent and sequential designs.
3. Verilog is used as an efficient HDL and supports four values, logical '0', logical '1', high impedance 'z' and unknown 'x'.
4. Verilog uses concurrent and sequential statement. Verilog HDL supports different operators to perform logical and arithmetic operations.
5. Verilog is used for both design and verification of digital logic.
6. Verilog is case sensitive and have synthesizable and non-synthesizable constructs.

# Chapter 2

## Combinational Logic Design (Part I)



An efficient RTL design engineer always works on the optimal design constraints and uses minimum number of logic gates. This chapter describes about the combinational Logic design and synthesizable Verilog RTL. Also deals with the practical and real life scenarios, useful while implementing combinational designs.

**Abstract** This chapter describes the use of Verilog HDL to code the combinational logic design and covers the small gate count designs. The chapter is organized in such a way that it can give the practical synthesizable Verilog HDL understanding with key practical scenarios and applications. The synthesizable Verilog HDL is described for the required functionality and the synthesized logic is explained for practical understanding. This chapter is useful to build the practical expertise to code the combinational designs using synthesizable Verilog constructs.

**Keywords** Logic gates • NOT • AND • NAND • OR • NOR • EXOR • EXNOR • Buffer • Adder • Subtractor • Gray • Binary • Code-conversion • Blocking assignment • Continuous assignment • Procedural lock • Always • Tri state • Two’s compliment

### 2.1 Introduction to Combinational Logic

Combinational logic is implemented by using the logic gates and in the combinational logic, output is the function of present input. The goal of a designer is always to implement the logic using minimum number of logic gates or logic cells. Minimization techniques are K-map, Boolean algebra, Shannon’s expansion theorems, and hyper planes. The thought process of a designer should be such that; the

design should have the optimal performance with lesser area density. The area minimization techniques have an important role in the design of combinational logic or functions. In the present scenario, designs are very complex; the design functionality is described using the hardware description language Verilog. The subsequent section focuses on the use of Verilog RTL to describe the combinational design.

## 2.2 Logic Gates and Synthesizable RTL

This section discusses about the logic gates and the synthesizable Verilog RTL.

### 2.2.1 *NOT or Invert Logic*

NOT logic complements the input. NOT logic is also called as inverter. Synthesizable RTL is shown in the Example 2.1. The truth table of NOT logic is shown in the Table 2.1.

Synthesized NOT logic is shown in the Fig. 2.1, input port of NOT logic gate is named as 'a\_in' and output as 'y\_out.'

### 2.2.2 *Two-Input OR Logic*

OR logic generates output as logical '1' when one of the input is logical '1.' Synthesizable RTL is shown in the Example 2.2. The truth table of OR logic is shown in the Table 2.2.

Synthesized OR logic is shown in the Fig. 2.2, input ports of OR logic gate are named as 'a\_in,' 'b\_in,' and output as 'y\_out'.

### 2.2.3 *Two-Input NOR Logic*

NOR logic is the opposite or complement of the OR logic. Synthesizable RTL is shown in the Example 2.3. The truth table of NOR logic is shown in the Table 2.3. NOR is universal logic gate, Bubbled AND is NOR and it is DeMorgans's theorem.

Synthesized NOR logic is shown in the Fig. 2.3, input ports of NOR logic gates are named as 'a\_in,' 'b\_in,' and output as 'y\_out.'

```
// Verilog RTL code for Not or Invert Logic

module not_logic (a_in,y_out);

input a_in;

output y_out;

// Functionality of design

assign y_out = ~a_in;

endmodule

// Verilog RTL code for Not or Invert Logic using procedural block

module not_logic (a_in,y_out);

input a_in;

output y_out;

reg y_out;

// Functionality of design

always@(a_in)

y_out = ~a_in;

endmodule
```

assign is used for continuous assignment to describe the combinational logic.

Continuous assignments are used to assign values to nets. Here net type is wire.

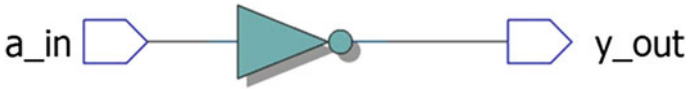
'always' block is procedural block and executes when there is an event on 'a\_in' input.

An output 'y\_out' is updated with the not of 'a\_in' input. As 'y\_out' is used inside the procedural block it is declared as 'reg'.

**Example 2.1** Synthesizable Verilog code for NOT logic

**Table 2.1** Truth table for NOT logic

a_in	y_out
0	1
1	0



**Fig. 2.1** Synthesized NOT logic

```

//verilog RTL code for the OR Logic
module or_logic (a_in, b_in, y_out);
input a_in;
input b_in;
output y_out;
reg y_out;
always@ (a_in or b_in)
begin
    if (a_in==0 && b_in==0)
        y_out = 1'b0;
    else
        y_out = 1'b1;
end
endmodule
    
```

Procedural block executes when there is event on either 'a\_in' or 'b\_in'.

Output is assigned to logic '0' when 'a\_in', 'b\_in' both are zero.

Output y\_out is assigned to logical '1' when one of the input 'a\_in' or 'b\_in' is either logical one.

OR logic can be expressed using  
 assign y\_out = a\_in | b\_in;  
 where y\_out is wire.

**Example 2.2** Synthesizable Verilog code for two-input OR logic. *Note* While describing the design functionality; make sure that all the input ports are listed in the sensitivity list. Missing required signals from sensitivity list will create simulation and synthesis mismatch and will be discussed in Chap. 3

**Table 2.2** Truth table for two-input OR logic

a_in	b_in	y_out
0	0	0
0	1	1
1	0	1
1	1	1

**Fig. 2.2** Synthesized two-input OR logic



```

//verilog RTL code for the NOR Logic
module nor_logic (a_in, b_in, y_out);

input a_in;

input b_in;

output y_out;

reg y_out;

always@ (a_in or b_in)

begin

    if (a_in==0 && b_in==0)

        y_out = 1'b1;

    else

        y_out = 1'b0;

end

endmodule
    
```

Procedural block executes when there is event on either 'a\_in' or 'b\_in'.

Output is assigned to logic '1' when 'a\_in', 'b\_in' both are zero.

Output y\_out is assigned to logical '0' when one of the input 'a\_in' or 'b\_in' is either logical one.

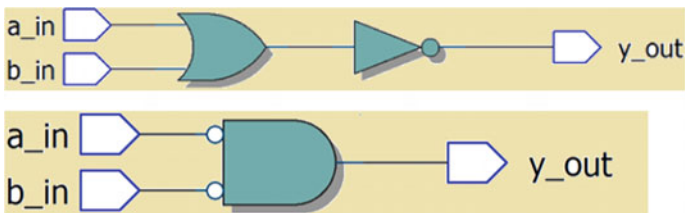
NOR logic can be expressed using :

assign y\_out =~ (a\_in | b\_in);  
 where y\_out is wire.

**Example 2.3** Synthesizable Verilog code for NOR logic

**Table 2.3** Truth table for two-input NOR logic

a_in	b_in	y_out
0	0	1
0	1	0
1	0	0
1	1	0



**Fig. 2.3** Synthesized two-input NOR logic



### 2.2.4 Two-Input AND Logic

AND logic generates an output as logical '1' when both the inputs 'a,' 'b,' are logical '1.' Synthesizable RTL is shown in the Example 2.4. The truth table of AND logic is shown in the Table 2.4.

```
//verilog RTL code for the AND Logic
module and_logic (a_in, b_in, y_out);

input a_in;

input b_in;

output y_out;

reg y_out;

always@ (a_in or b_in)

begin

    if (a_in==1 && b_in==1)

        y_out = 1'b1;

    else

        y_out = 1'b0;

end

endmodule
```

Procedural block executes when there is event on either 'a\_in' or 'b\_in'.

Output is assigned to logic '1' when 'a\_in', 'b\_in' both are one.

Output y\_out is assigned to logical '0' when one of the input 'a\_in' or 'b\_in' is either logical zero.

AND logic can be expressed using :

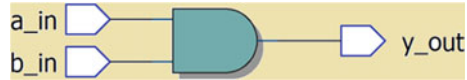
assign y\_out =(a\_in & b\_in) ;  
where y\_out is wire.

**Example 2.4** Synthesizable Verilog code for two-input AND logic. *Note* AND gate is visualized as a series of two switches and used in programmable logic devices (PLD) as one of the element to realize the required logic. Programmable AND plane can be created using the AND logic gates as primary elements having feature as programmable inputs

**Table 2.4** Truth table for two-input AND logic

a_in	b_in	y_out
0	0	0
0	1	0
1	0	0
1	1	1

**Fig. 2.4** Synthesized two-input AND logic



Synthesized two-input AND logic is shown in the Fig. 2.4, input ports of AND logic gate are named as ‘a\_in,’ ‘b\_in,’ and output as ‘y\_out.’

### 2.2.5 Two-Input NAND Logic

NAND logic is the opposite or complement of the AND logic. Synthesizable RTL is shown in the Example 2.5. The truth table of NAND logic is shown in the Table 2.5.

```

//verilog RTL code for the NAND Logic
module nand_logic (a_in, b_in, y_out);

input a_in;

input b_in;

output y_out;

reg y_out;

always@ (a_in or b_in)

begin

    if (a_in==1 && b_in==1)

        y_out = 1'b0;

    else

        y_out = 1'b1;

end

endmodule
    
```

Procedural block executes when there is event on either ‘a\_in’ or ‘b\_in’.

Output is assigned to logic ‘0’ when ‘a\_in’, ‘b\_in’ both are one.

Output y\_out is assigned to logical ‘1’ when one of the input ‘a\_in’ or ‘b\_in’ is either logical zero.

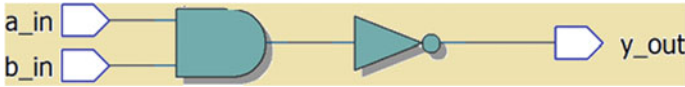
NAND logic can be expressed using :

assign y\_out =~(a\_in & b\_in) ;  
 where y\_out is wire.

**Example 2.5** Synthesized Verilog RTL for two-input NAND Logic. *Note* NAND logic is also treated as universal logic. Using NAND logic, all possible logic functions can be realized. NAND logic is used to implement the storage elements like latches or flip-flops and also to realize combinational functions. According to DeMorgan’s theorem the bubbled OR is equivalent to NAND

**Table 2.5** Truth table for two-input NAND logic

a_in	b_in	y_out
0	0	1
0	1	1
1	0	1
1	1	0

**Fig. 2.5** Synthesized two-input NAND logic

Synthesized NAND logic is shown in the Fig. 2.5, input ports of NAND logic gate is named as ‘a\_in,’ ‘b\_in,’ and output as ‘y\_out.’

### 2.2.6 Two-Input XOR Logic

Two-input XOR is called as exclusive OR logic and generates output as logical ‘1,’ when both inputs are not equal. Synthesizable RTL is shown in the Example 2.6. The truth table of XOR logic is shown in the Table 2.6.

Synthesized two-input XOR logic is shown in the Fig. 2.6; input ports of XOR logic gate are named as ‘a\_in,’ ‘b\_in,’ and output as ‘y\_out.’

If XOR cell or gate is not available in the library then XOR logic is realized using AND-OR-Invert or using minimum number of NAND gates.

### 2.2.7 Two-Input XNOR Logic

Two-input XNOR is called as exclusive NOR logic and generates output as logical ‘1’ when both the inputs are equal. XNOR is opposite or complement of XOR logic. Synthesizable RTL for XNOR is shown in the Example 2.7. The truth table of XNOR logic is shown in the Table 2.7.

Synthesized XNOR logic is shown in the Fig. 2.7, input ports of XNOR logic gate are named as ‘a\_in,’ ‘b\_in,’ and output as ‘y\_out’.

If XNOR cell is not available in the library then XNOR logic is realized using AND-OR-Invert or using minimum number of NAND or NOR gates. Minimum five two input NAND gates are required to realize the 2 input XNOR gate.

```

//verilog RTL code for the XOR Logic
module xor_logic (a_in, b_in, y_out);

input a_in;

input b_in;

output y_out;

reg y_out;

always@ (a_in or b_in)

begin
    if (a_in!=b_in)
        y_out = 1'b1;
    else
        y_out = 1'b0;
end

endmodule
    
```

Procedural block executes when there is event on either 'a\_in' or 'b\_in'.

Output is assigned to logic '0' when 'a\_in', 'b\_in' both are having same logic value.

Output y\_out is assigned to logical '1' when both of the inputs 'a\_in' or 'b\_in' are having different logic values.

XOR logic can be expressed using :

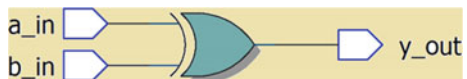
assign y\_out =(a\_in ^ b\_in) ;  
where y\_out is wire.

**Example 2.6** Synthesizable Verilog code for two-input XOR logic. *Note* XOR gate can be implemented using two-input NAND gates. The number of two-input NAND gates required to implement two-input XOR gate are equal to 4. XOR gates are used to implement arithmetic operations such as addition and subtraction

**Table 2.6** Truth table for two-input XOR logic

a_in	b_in	y_out
0	0	0
0	1	1
1	0	1
1	1	0

**Fig. 2.6** Synthesized two-input XOR logic



```

//verilog RTL code for the XNOR Logic
module xnor_logic (a_in, b_in, y_out);

input a_in;

input b_in;

output y_out;

reg y_out;

always@ (a_in or b_in)

begin

    if (a_in!=b_in)

        y_out = 1'b0;

    else

        y_out = 1'b1;

end

endmodule
    
```

Procedural block executes when there is event on either 'a\_in' or 'b\_in'.

Output is assigned to logic '1' when 'a\_in', 'b\_in' both are having same logic value.

Output y\_out is assigned to logical '0' when both of the inputs 'a\_in' or 'b\_in' are having different logic values.

XNOR logic can be expressed using :

assign y\_out =~ (a\_in ^ b\_in);  
 where y\_out is wire.

**Example 2.7** Synthesizable Verilog code for XNOR logic

**Table 2.7** Truth table for XNOR logic

a_in	b_in	y_out
0	0	1
0	1	0
1	0	0
1	1	1

**Fig. 2.7** Synthesized XNOR logic



### 2.2.8 Tri-state Logic

Tri-state has three logic states namely, logical ‘0,’ logical ‘1,’ and high impedance ‘z.’ Synthesizable RTL is shown in the Example 2.8. The truth table of tri-state buffer logic is shown in the Table 2.8.

Synthesized tri-state logic is shown in the Fig. 2.8, input port of tri-state NOT logic is named as ‘data\_in,’ enable input as ‘enable’ and output as ‘data\_out.’

```
// Verilog RTL code for tri-state logic

module tri_sate_logic (data_in,enable, data_out);

input [3:0] data_in;

input enable;

output [3:0] data_out;

reg [3:0] data_out;

// Functionality of design

always@(data_in, enable)

if (enable)

data_out = data_in;

else

data_out= 4'bzzzz;

endmodule
```



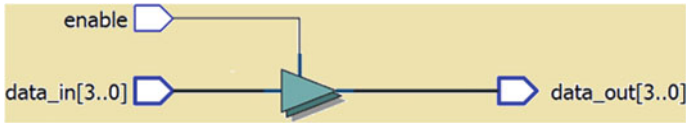
‘always’ block is procedural block and executes when there is an event on ‘data\_in’ input or enable.

An output ‘data\_out’ is updated with the current value of ‘data\_in’ for ‘enable=1’. For ‘enable=0’ an output of tri-state logic is high impedance.

**Example 2.8** Synthesizable Verilog code for tri-state logic. *Note* Avoid use of tri-state logic while developing the RTL. Tri state is difficult to test. Instead of tri-state logic, it is recommended to use multiplexers to develop the logic with enable

**Table 2.8** Truth table for tri-state logic

enable	data_in	data_out
1	0000	0000
1	1111	1111
0	xxxx	zzzz

**Fig. 2.8** Synthesized tri-state NOT logic

## 2.3 Arithmetic Circuits

Arithmetic operations such as addition and subtraction has an important role in the efficient design of processor logic. Arithmetic logic unit (ALU) of any processor can be designed to perform the addition, subtraction, increment, decrement operations. The arithmetic designs are described by the RTL Verilog code to achieve the optimal area and less critical path. This section describes the important logic blocks to perform arithmetic operations with the equivalent Verilog RTL description.

### 2.3.1 Adder

Adders are used to perform the binary addition of two binary numbers. Adders are used for signed or unsigned addition operations.

#### 2.3.1.1 Half Adder

Half adder has two, one-bit inputs 'a\_in,' 'b\_in' and generates two, one-bit outputs 'sum\_out,' 'carry\_out.' Where 'sum\_out' is the summation or addition output and 'carry\_out' is the carry output. Table 2.9 is the truth table for half adder and RTL is described in the Example 2.9.

**Table 2.9** Truth table for half adder

a_in	b_in	sum_out	carry_out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

```

//Verilog RTL code for half adder

module half_adder ( a_in, b_in, sum_out,carry_out);

input a_in;

input b_in;

output sum_out;

output carry_out;

wire sum_out;

wire carry_out;

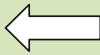
assign sum_out = a_in ^ b_in;

assign carry_out = a_in & b_in;

endmodule
    
```

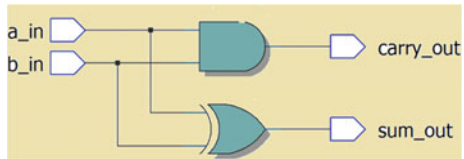
Output sum\_out is assigned as XOR of 'a\_in', 'b\_in'. XOR is summation operation.

Output carry\_out is assigned as AND of 'a\_in', 'b\_in'. Carry logic is AND operation.



**Example 2.9** Synthesizable RTL code for half adder. *Note* Half adders are used as basic component to perform the addition. Full adder logic circuits are designed using the instantiation of half adders as components

**Fig. 2.9** Synthesized half adder



Synthesized half adder is shown in the Fig. 2.9, input ports of half adder are named as 'a\_in,' 'b\_in,' and output as 'sum\_out,' 'carry\_out.'

**2.3.1.2 Full Adder**

Full adders are used to perform addition on three, one-bit binary inputs. Consider three, one-bit binary numbers named as 'a\_in,' 'b\_in,' 'c\_in' and one-bit binary outputs as 'sum\_out,' 'carry\_out.' Table 2.10 is the truth table for full adder and RTL is described in the Example 2.10.



**Table 2.10** Truth table for full adder

c_in	a_in	b_in	sum_out	carry_out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

```
//Verilog RTL code for one bit full adder
```

```
module full_adder ( a_in, b_in, c_in, sum_out, carry_out);
```

```
input a_in;
```

```
input b_in;
```

```
input c_in;
```

```
output sum_out;
```

```
output carry_out;
```

```
wire sum_out;
```

```
wire carry_out;
```

```
assign {carry_out, sum_out} = a_in + b_in + c_in;
```

```
endmodule
```

'assign' is continuous assignment statement and used to assign the one bit sum output 'sum\_out' and carry output 'carry\_out' after performing addition operation.

**Example 2.10** Synthesizable Verilog code for full adder. *Note* Full adder consumes more area so it is highly recommended to implement the adder logic using multiplexers

Synthesized full adder is shown in the Fig. 2.10, input ports of full adder are named as 'a\_in,' 'b\_in,' 'c\_in' and output as 'sum\_out' 'carry\_out.'

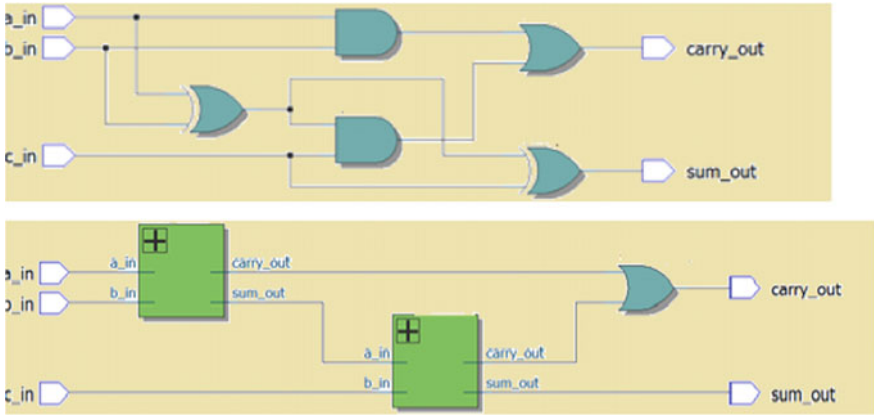


Fig. 2.10 Synthesized full adder

### 2.3.2 Subtractor

Subtractors are used to perform the binary subtraction of two binary numbers. This section describes about the half and full subtractors.

#### 2.3.2.1 Half Subtractor

Half subtractor has two, one-bit inputs ‘a,’ ‘b’ and generates two one-bit outputs ‘d,’ ‘bor’. Where ‘d’ is difference output and ‘bor’ is borrow output. Table 2.11 is the truth table for half subtractor and RTL is described in the Example 2.11.

Synthesized half subtractor is shown in the Fig. 2.11, input ports of half adder are named as ‘a,’ ‘b,’ and output as ‘d,’ ‘bor.’

#### 2.3.2.2 Full Subtractor

Full subtractors are used to perform subtraction of three, one-bit binary inputs. Consider three, one-bit numbers named as ‘a,’ ‘b,’ ‘c’ and one-bit binary outputs as ‘d,’ ‘bor.’ Table 2.12 is the truth table description for full subtractor and RTL is described in the Example 2.12 and Fig. 2.12.

Table 2.11 Truth table for half subtractor

a	b	d	bor
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

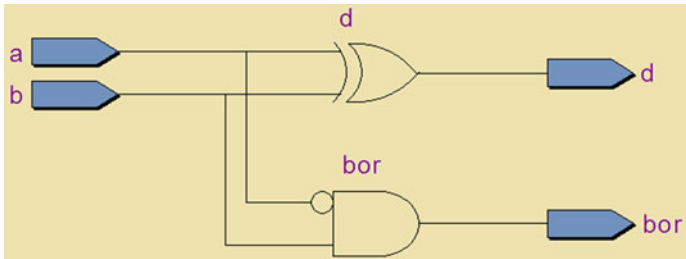
```
// Verilog RTL code for half subtractor
module half_subtractor (a, b, d, bor);
input a;
input b;
output d;
output bor;
wire d;
wire bor;
assign d = a ^ b;
assign bor = (~a & b);
endmodule
```

Output 'd' is assigned as XOR of 'a', 'b'. XOR is used to generate output.

Output 'bor' is assigned as complement of 'a' AND 'b'



**Example 2.11** Synthesizable Verilog code for half subtractor. *Note* Half subtractors are used as basic component to perform the binary subtractions. Full subtractor logic circuits are designed using the instantiation of half subtractors as components



**Fig. 2.11** Synthesized half subtractor

**Table 2.12** Truth table for full subtractor

c	a	b	d	bor
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

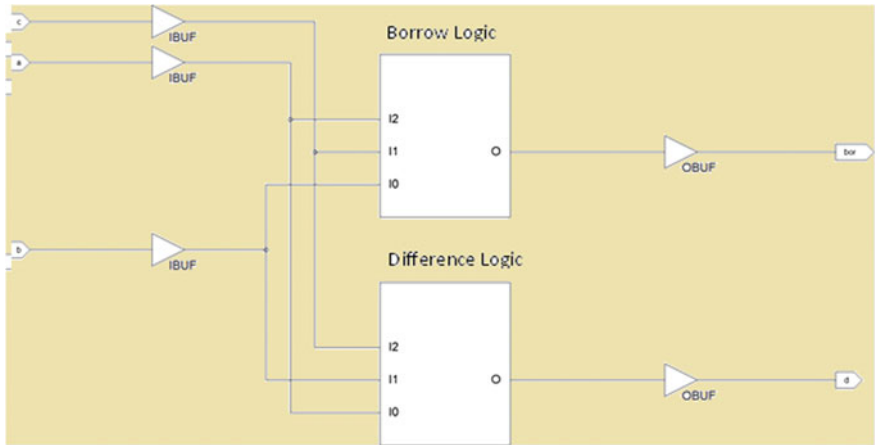
```

module full_subtractor (a, b, c, d, bor);
input a;
input b;
input c;
output d;
output bor;
reg [1:0] temp;
always @ (a or b or c)
begin
temp = a - b - c;
end
assign d = temp[0];
assign bor = temp[1];
endmodule
    
```

'temp' is two bit temporary variable used to hold the data. 'temp' is declared as reg as it is used in the always block.

'In the expression subtraction of 'a', 'b', 'c' is assigned to 'temp' using binary arithmetic subtraction ('-') operator.

**Example 2.12** Synthesizable Verilog code for full subtractor. *Note* It is recommended to use the full adder to perform the subtraction operation. Subtraction is performed using two's complement addition



**Fig. 2.12** Synthesized full subtractor

Synthesized full subtractor is shown in the Fig. 2.12 input ports of full subtractor are named as 'a,' 'b,' 'c' and output as 'd,' 'bor.'

### 2.3.3 Multi-bit Adders and Subtractors

Multi-bit adders and subtractors are used in the design of arithmetic units for the processors. The logic density depends upon the number of input bits of adder or subtractor.

#### 2.3.3.1 Four-Bit Full Adder

Many practical designs use multi-bit adders and subtractors. It is the industrial practice to use basic component as full adder to perform the addition operation. For example, if designer needs to implement the four-bit design logic of an adder, then four full adders are required. As shown in the Example 2.13, addition is performed on two, four-bit binary numbers 'A,' 'B.' The final result is four-bit addition and output at 'S.' Carry input is Ci and carry output is Co.

Synthesized four-bit adder is shown in the Fig. 2.13, input ports of four-bit adder are named as 'A,' 'B,' 'Ci,' and output as 'S,' 'Co.'

```
// Verilog RTL code for 4 bit full adder
module adder_4bit (A,B,Ci,S,Co);
parameter data_size =4;
input [data_size-1:0] A;
input [data_size-1:0] B;
input Ci;
output [data_size-1:0] S;
output Co;
// Functionality of design
assign {Co, S} = A + B + Ci;
endmodule
```

parameter is used to define the data\_size according to the given design specifications.

assign is used for assigning the carry output 'Co' and 4-bit sum output 'S' after addition.

**Example 2.13** Synthesizable Verilog code for four-bit adder. *Note* Four-bit addition operation uses four full adders. Depending on signed or unsigned addition requirements the Verilog code can be modified

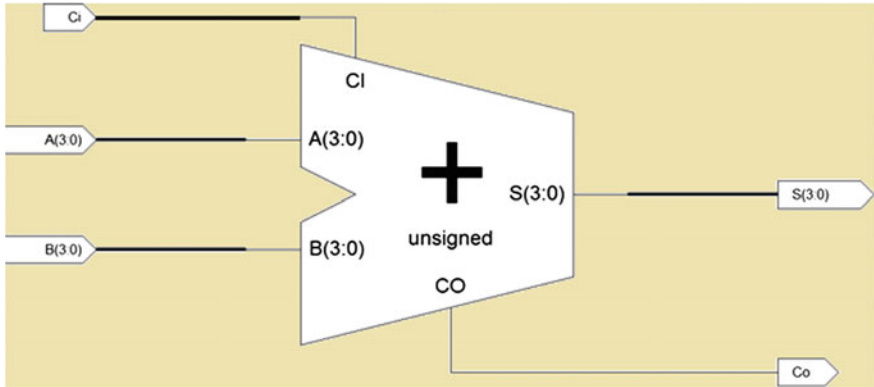


Fig. 2.13 Synthesized four-bit adder

**2.3.3.2 Four-Bit Adder and Subtractor**

Design of addition and subtraction can be accomplished using the adders only. Subtraction can be performed using two’s complement addition. For example consider the scenario shown in the Table 2.13.

Synthesized four-bit adder/subtractor is shown in the Fig. 2.14, for Example 2.14, input ports of four-bit adder/subtractor are named as ‘A,’ ‘B,’ ‘Ci,’ and output as ‘S,’ ‘Co.’ When control input SUB is equal to logic ‘0’ then it performs the addition and for control input SUB is equal to logic ‘1’ it performs the subtraction which is 2’s complement addition.

**Table 2.13** Operational table for adder subtractor

Operation	Description	Expression
Addition	Unsigned addition of A, B	$A + B + 0$
Subtraction	Unsigned subtraction of A, B	$A - B = A + \sim B + 1$

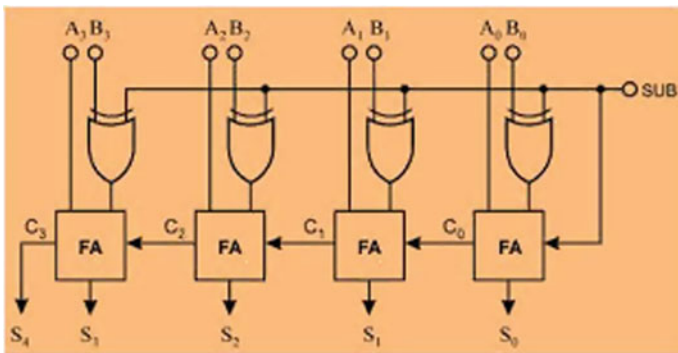


Fig. 2.14 Synthesized four-bit adder/subtractor

```
// Verilog RTL code for 4 bit full adder subtractor
```

```
module adder_subtractor_4bit (A,B,Ci,S,Co);
```

```
parameter data_size =4;
```

```
input [data_size-1:0] A;
```

```
input [data_size-1:0] B;
```

```
input Ci;
```

```
output [data_size-1:0 ] S;
```

```
output Co;
```

```
reg [data_size-1:0] S;
```

```
reg Co;
```

```
// Functionality of design
```

```
always @ ( A or B or Ci)
```

```
if (Ci)
```

```
{Co,S}= A - B;
```

```
else
```

```
{Co,S}=A + B;
```

```
endmodule
```

parameter is used to define the data\_size according for the given design specifications.

For Ci='1' S= A-B else  
S=A+B

**Example 2.14** Synthesizable Verilog code for four-bit adder and subtractor. *Note* Consider SUB control,input as Ci and S4 as Co in the synthesized logic. Here, the resource used is binary full adder to perform both the additions and subtractions. Subtraction operation is performed using adders only. Resource sharing and resource utilization are to be discussed in the Chap. 3

### 2.3.4 Comparators and Parity Detectors

In most of the practical scenarios; comparators are used to compare the equality of two binary numbers. Parity detectors are used to compute the even or odd parity for the given binary number. It becomes very essential for the design engineer to have the better understanding of this.

#### 2.3.4.1 Binary Comparators

These are used to compare the two binary numbers. As discussed earlier Verilog supports four value logic and they are logical '0,' logical '1,' don't care 'x' and high impedance 'z.' Verilog supports logical equality operator (==) and inequality

**Table 2.14** Operational table for comparator


Condition	Description	Verilog expression
A==B	Assign output as XOR of A, B	A ^ B
A!=B	Assign output as AND of A, B	A & B

operator (!=), and these are used to describe the comparison of two numbers. These operators are used in the Verilog Synthesizable RTL code.

For example consider the operational Table 2.14. As shown in the table; when A, B both are equal then output ‘Y’ is assigned to XOR of ‘A,’ ‘B’ and for unequal case output ‘Y’ is assigned to AND of ‘A,’ ‘B’ (Example 2.15).

Synthesized equivalent block representation is shown in the Fig. 2.15 (Example 2.15).

```
// Verilog RTL code for 1-bit comparator
module comparator (A,B,Y);
input A;
input B;
output Y;
reg Y;
// Functionality of design
always @ ( A or B)
begin
    if ( A==B)
        Y = A ^ B;
    else
        Y = A & B;
end
endmodule
```



Equality operator ‘==’ is used for comparison. For ‘x’ or ‘z’ inputs the result of logical comparison is always false. If ‘A’ or ‘B’ becomes either ‘x’ or ‘z’ then the result will be A & B as

**Example 2.15** Synthesizable Verilog code for 1-bit comparator. *Note* Logical equality and inequality operators are used in the synthesizable RTL code and for any of the operands are ‘x’ or ‘z’ comparison is false



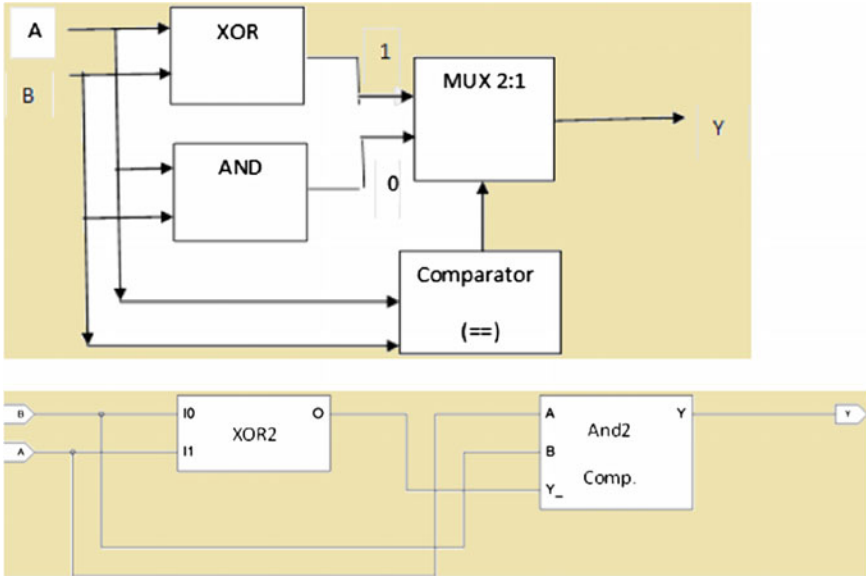


Fig. 2.15 Synthesized equality comparator

### 2.3.4.2 Parity Detector

Parity detectors are used to detect the even or odd parity for the binary number string. For even number of 1’s, the output required is logical ‘0’ and for odd number of 1’s the output required is logical ‘1,’ then the RTL Verilog can be described as shown in the Example 2.16.

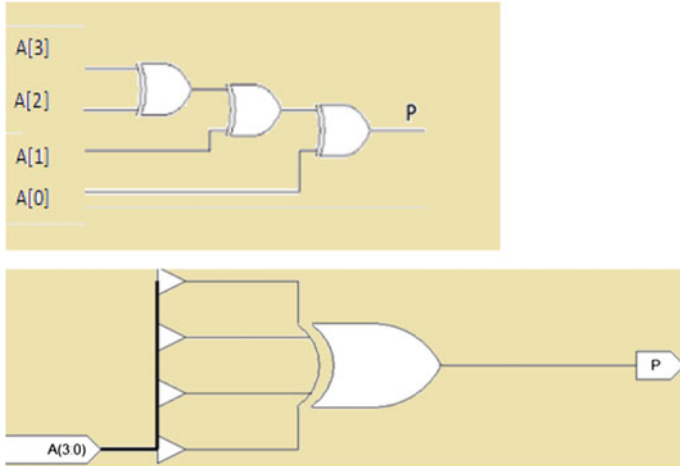
```
// Verilog RTL code ffor parity detector
module parity_logic (A, P);
input [3:0] A;
output P;
wire P;
// Functionality of design
assign P = ( A[3] ^ A[2] ^ A[1] ^ A[0] );
endmodule
```

XOR (^) operator is used to detect the parity . The assign statement assigns the output ‘P’ to logical ‘1’ for odd number of 1’s and for even number of 1’s ouput is equal to logical ‘0’

Example 2.16 Synthesizable Verilog code for parity detector. Note Parity detectors are used in many of the DSP applications and an integral module for encryption engines

**Table 2.15** Operational table for parity detector

Condition	Description
Odd 1's	Assign output as logical 1
Even 1's	Assign output as logical zero



**Fig. 2.16** Synthesized parity detector

The operational table for the parity detector is shown below in Table 2.15. For odd number of 1's the output is logical '1' and for even number of 1's output is assigned as logical '0'.

Synthesized equivalent block representation is shown in the Fig. 2.16.

### 2.3.5 Code Converters

This section deals with the commonly used code converters in the design. As name itself indicates the code converters are used to convert the code from one number system to another number system. In the practical scenarios, binary to gray and gray to binary converters are used.

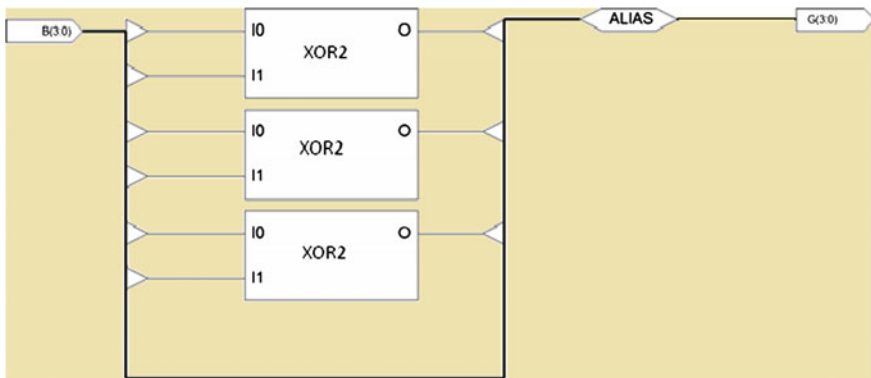
#### 2.3.5.1 Binary to Gray Code Converter

Base of binary number system is 2, for any multi-bit binary number one or more than one bit changes at a time. In gray code, only one bit changes at a time.

The RTL description of four-bit binary to gray code conversion is described in Example 2.17.

```
// Verilog RTL code for 4-bit binary to Gray converterr
module binary_to_gray (B, G);
input [3:0] B;
output [3:0] G;
// Functionality of design
assign G[3] = B[3];
assign G[2] = B[3] ^ B[2];
assign G[1] = B[2] ^ B[1];
assign G[0] = B[1] ^ B[0];
endmodule
```

**Example 2.17** Synthesizable Verilog code for four-bit binary to gray code converter. *Note* Gray codes are used in the multiple clock domain designs to transfer the control information from one of the clock domain to another clock domain



**Fig. 2.17** Synthesized four-bit binary to gray converter

Synthesized equivalent block representation is shown in Fig. 2.17.

### 2.3.5.2 Gray to Binary Code Converter

Gray to binary code converter is opposite of that of binary to gray and the RTL description of four-bit gray to binary code conversion described in Example 2.18.

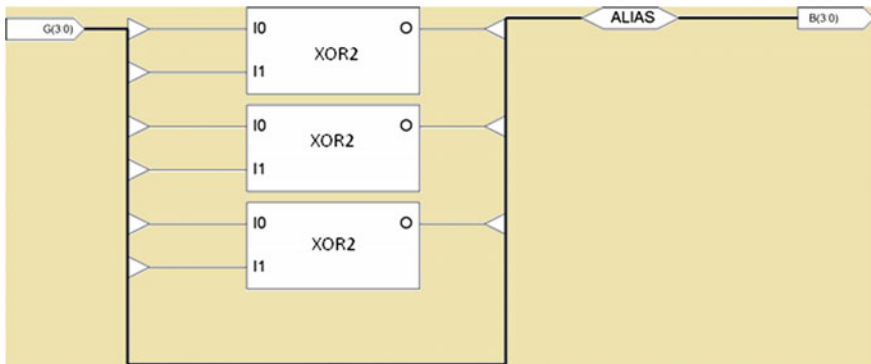
Synthesized equivalent block representation is shown the Fig. 2.18.

```
// Verilog RTL code for 4-bit gray to binary converter  
module gray_to_binary (G, B);  
input [3:0] G;  
output [3:0] B;  
  
// Functionality of design  
assign B[3] = G[3];  
assign B[2] = G[3] ^ G[2];  
assign B[1] = G[3] ^ G[2] ^ G[1];  
assign B[0] = G[3] ^ G[2] ^ G[1] ^ G[0];  
endmodule
```

Gray to Binary code uses the XOR operator to realize the equivalent logic.



**Example 2.18** Synthesizable Verilog code for four-bit gray to binary code converter. *Note* Gray codes are used in the gray counter implementation and also in the error correcting mechanism



**Fig. 2.18** Synthesized four-bit gray to binary converter

## 2.4 Summary

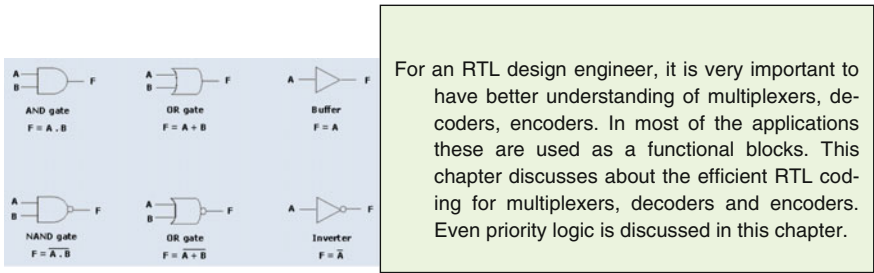
As discussed already in this chapter; following are the important points need to be considered while implementing combinational logic RTL.

1. Use minimum area by sharing the arithmetic resources.
2. Use all the required signals in the sensitivity list to avoid simulation and synthesis mismatch.
3. Avoid use of tri-state logic and implement the logic required using multiplexers with proper enable circuit.

4. Verilog supports four value logic and they are logical '0,' logical '1,' don't care 'x,' high impedance 'z.'
  - Use less number of adders in design. Adders can be implemented using multiplexers.
  - NAND and NOR are universal logic gates and used to implement any combinational or sequential logic.

# Chapter 3

## Combinational Logic Design (Part II)



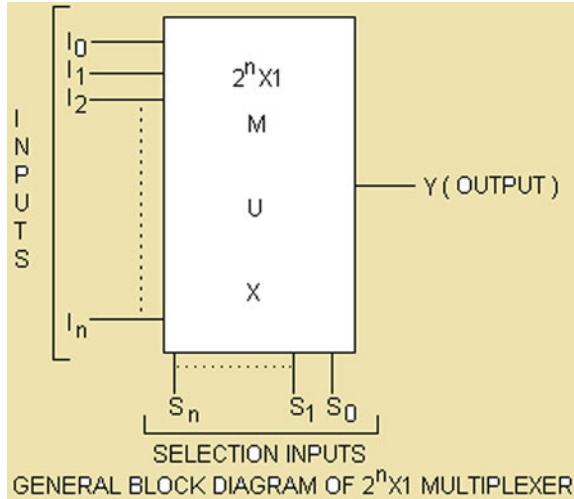
**Abstract** This chapter describes the complex combinational logic designs and covers the detail and practical oriented scenarios while describing the multiplexers, decoders, encoders, and priority encoders. The use of constructs like “if-else,” “case,” and continuous assignment “assign” are described in detail with the meaningful practical examples. The main focus of this chapter is to describe the design functionality with the synthesizable logic. Even this chapter focuses on the key practical issues need to be tackled while describing the Verilog HDL.

**Keywords** MUX · Decoder · Encoder · Priority logic · Parallel logic · Nested statement · Concurrent execution · Do not care · Priority · Inference · Simulation · Synthesis · Gate level · Block level

### 3.1 Multiplexers

Multiplexers are used to select one of the input from many. Multiplexers are also called as universal logic and terminology used in the practical world is MUX. By using the suitable multiplexers any of combinational logic function can be realized. Multiplexers are used as selection logic in ASIC and FPGA-based designs. Multiplexer consumes lesser area as compared to adders and most of the time MUX is used to implement arithmetic components such as adders and subtractors.

**Fig. 3.1** Block diagram of  $n:1$  MUX



The block diagram of  $n:1$  MUX is shown in Fig. 3.1 and it consists of ‘ $n$ ’ input lines, ‘ $m$ ’ select lines, and one output line. Input lines are denoted as “ $i[0], i[1], \dots, i[n - 1]$ ,” select lines by “ $s[0], s[1], \dots, s[m - 1]$ ,” and output line by ‘ $y$ ’.

As shown in Fig. 3.1 multiplexer has ‘ $n$ ’ input lines, ‘ $m$ ’ select lines, and single output line. Relation between the input lines and select lines is given by  $n = 2^m$ . For example; for 4:1 MUX input lines are four so  $m = \log_2 n$  that is select lines equal to two.

### 3.1.1 Multiplexer as Universal Logic

As discussed earlier multiplexer is treated as universal logic as all combinational logic functions can be realized using MUX.

#### 3.1.1.1 2:1 MUX

A 2:1 MUX has two input lines, one select line and one output line. When ‘ $s$ ’ input is logical ‘0’ output ‘ $y$ ’ is assigned to ‘ $a$ ’ and output is assigned to ‘ $b$ ’ for ‘ $s$ ’ equal to logical ‘1’. Table 3.1 describes the truth table of 2:1 MUX and Example 3.1 is synthesizable RTL for 2:1 multiplexer.

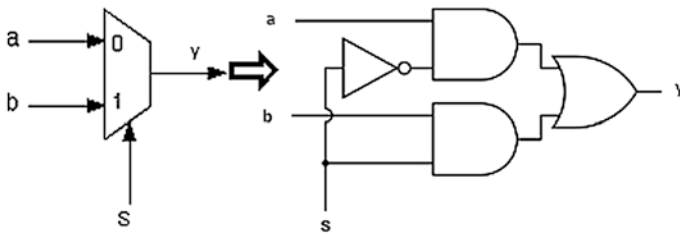
**Table 3.1** Truth table for 2:1 MUX

$s$	$y$
0	$a$
1	$b$

```
// Verilog RTL code for 2:1 Multiplexer  
module mux_2to1 (a, b, s, y);  
input a;  
input b;  
input s;  
output y;  
// Functionality of design  
assign y = (s) ? b : a;  
endmodule
```

'assign' is used for continuous assignment at 'y'. The conditional expression  $y = (s) ? b : a;$  is equivalent to (condition) ? (input1) : (input0);  
When  $s = '1'$   $y = b;$  when  $s = '0'$   $y = a$

**Example 3.1** Synthesizable Verilog code for 2:1 MUX. *Note* Conditional assignments are used to select from many inputs so infers the multiplexer



**Fig. 3.2** Synthesized 2:1 MUX. *Note* A 2:1 multiplexer symbolic representation is used to describe the implementation of higher complexity multiplexers. Multiplexer is treated as universal logic. Using multiplexers all possible combinational logic can be realized

The synthesizable RTL is shown in Fig. 3.2.

The reason for using MUX as universal logic is due to it's easy to understand and simple structure. Figure 3.3 describes how 2:1 MUX is used to implement the two input XOR logic gate. Consider XOR logic gate has two inputs 'a', 'b' and output 'y'. The implementation of two input XOR logic gate using 2:1 MUX is shown in Fig. 3.3.

Let us discuss the other ways to describe the 2:1 MUX. There are different ways in which 2:1 MUX can be described. It can be described using "if-else" or by using "case," Example 3.2 describes synthesizable RTL using "if-else" and Example 3.3 describes synthesizable RTL using "case" statement.



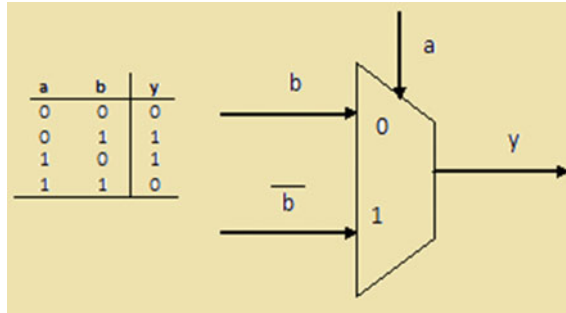


Fig. 3.3 Two input XOR logic using 2:1 MUX

```
// Verilog RTL code for 2:1 Multiplexer using if-else
module mux_2to1 (a, b, s, y);
input a, b, s;
output y;
// Functionality of design
always @ ( a or b or s)
if (s==1)
y=b;
else
y=a;
endmodule
```

If-else is used to describe the 2:1 MUX in the Verilog code.  
 When 's' is logical '1' then input 'b' is assigned to 'y' else input 'a' is assigned to 'y'

Example 3.2 Synthesizable Verilog code for 2:1 MUX using “if-else”

3.1.1.2 4:1 MUX Using “if-else”

Four as to one MUX has four input lines and single output lines. The 4:1 MUX has two select lines and used to select one of the inputs at a time. The truth table of 4:1 MUX is shown in Table 3.2 and Example 3.4 describes the synthesizable RTL for 4:1 MUX.

The equivalent synthesizable hardware inferred for the 4:1 MUX described in the Example 3.4 is shown in Fig. 3.4. As shown in Fig. 3.4 input 'a' has the highest priority as compared to other inputs. Input 'd' has least priority.

The hardware inferred in Fig. 3.4 is also generated by the Verilog RTL described in the Example 3.5.

```
// Verilog RTL code for 2:1 Multiplexer using case
module mux_2to1 (a, b, s, y);
input a, b, s;
output y;
// Functionality of design
always @ (a or b or s)
case (s)
1'b0 : y= a;
1'b1 : y=b;
endcase
endmodule
```

Case is used to describe the 2:1 MUX in the Verilog code.

When 's' is logical '1' then input 'b' is assigned to 'y' else input 'a' is assigned to 'y'

**Example 3.3** Synthesizable Verilog code for 2:1 MUX using “case”. Note “if-else” generates priority logic and “case” generates parallel logic. It is recommended to use “case” statement to describe MUX, decoders. It is recommended to use “if-else” to describe priority logic

**Table 3.2** Truth table of 4:1 MUX

S1	S0	y
0	0	a
0	1	b
1	0	c
1	1	d

### 3.1.1.3 4:1 MUX Using “case”

The 4:1 MUX is described by using the “case-endcase” construct and it is described in the Example 3.6. The synthesized hardware is shown in Fig. 3.5. As shown in Figure, “case-endcase” construct generates the parallel logic (Example 3.7).

### 3.1.1.4 4:1 MUX Using 2:1 MUX

The 4:1 MUX can be implemented by using 2:1 MUX and the equivalent representation is shown in Fig. 3.5.

As shown in Fig. 3.5; 4:1 MUX is implemented by using three 2:1 multiplexers. The Verilog RTL is described in the Example 3.8.

```
// Verilog RTL code for 4 :1 MUX
module Mux_4to1_priority(a,b,c,d,s1,s0,y);
input a;
input b;
input c;
input d;
input s1;
input s0;
output y;
reg y;
always @(a or b or c or d or s1 or s0)
begin
    if (s1==0 && s0==0)
        y=a;
    else if (s1==0 && s0==1)
        y=b;
    else if (s1==1 && s0==0)
        y=c;
    else
        y =d;
end
endmodule
```



The 4:1 MUX Verilog RTL with inputs declared as 'a', 'b', 'c', 'd' and select inputs as 's1', 's0'. The output from 4:1 MUX is 'y'



The functionality of 4:1 MUX is described using the nested if-else statement. Nested if-end implements the priority logic.

**Example 3.4** Synthesizable RTL for 4:1 MUX

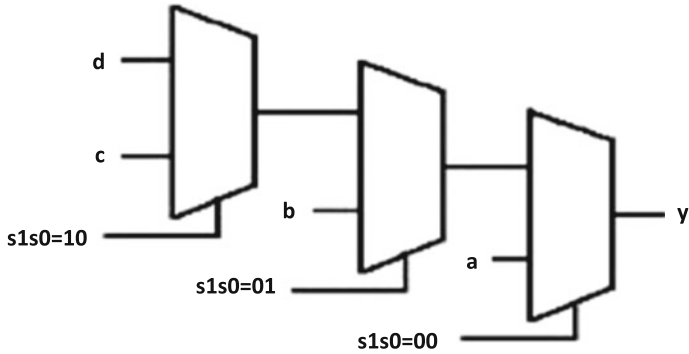


Fig. 3.4 Synthesized 4:1 MUX priority logic

*// Verilog RTL code for 4 :1 MUX*

**module** Mux\_4to1\_priority(in,sel,y);

**input** [3:0] in;

**input** [1:0] sel;

**output** y;

**reg** y;

**always** @(in or sel)

**begin**

**if** (sel==2'b00)

        y=in[0];

**else if** (sel==2'b01)

        y=in[1];

**else if** (sel==2'b10)

        y=in[2];

**else**

        y =in[3];

**end**

**endmodule**



The 4:1 MUX Verilog RTL with input declared as 4-bit vector 'in', select lines as 2-bit vector 'sel' and output as 'y'



Output 'y' is assigned to one of the inputs 'in[0], in[1], in[2], in[3]' depending on the status of select 'sel' line.

Example 3.5 Synthesizable RTL using if-else

```
// Verilog RTL code for 4 :1 MUX
module Mux_4to1_parallel(in,sel,y);
input [3:0] in;
input [1:0] sel;
output y;
reg y;
always @(in or sel)
begin
    case (sel)
        2'b00 : y=in[0];
        2'b01 : y=in[1];
        2'b10 : y=in[2];
        2'b11 : y=in[3];
    endcase
end
endmodule
```

The 4:1 MUX Verilog RTL with input declared as 4-bit vector 'in', select lines as 2-bit vector 'sel' and output as 'y'

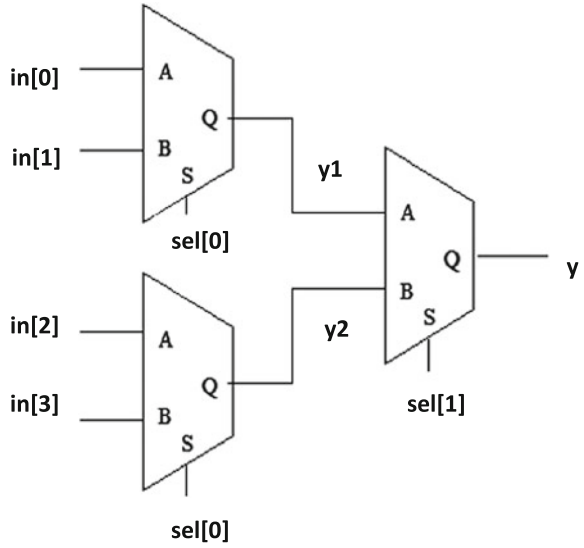


The 'case-endcase' construct is used to assign one of the input to 'y'. Output 'y' is assigned to one of the inputs 'in[0], in[1], in[2], in[3]' depending on the status of select on 'sel' line.

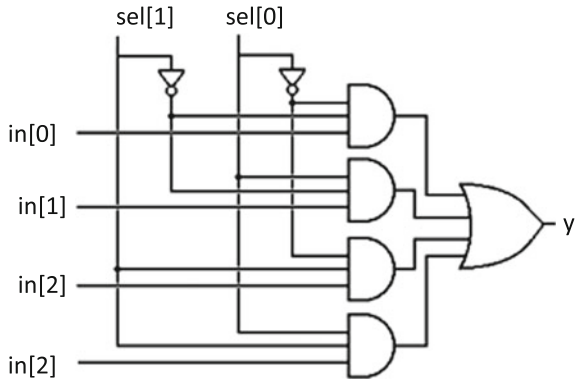


**Example 3.6** Synthesizable Verilog RTL using “case”

**Fig. 3.5** 4:1 Implementation using 2:1 MUX



**Example 3.7** Synthesized output for 4:1 MUX using “case”



```
//Implementation of 4:1 MUX using 2:1 MUX
module Mux_4to1_using2:1MUX(in,sel,y);
input [3:0] in;
input [1:0] sel;
output y;
reg y1,y2,y;
always @(in or sel[0])
begin
    case (sel[0])
        2'b0 : begin
            y1=in[0];
            y2=in[2];
        end
        2'b1 : begin
            y1=in[1];
            y2=in[3];
        end
    endcase
end
always @(y1 or y2 or sel[1])
begin
    case (sel[1])
        1'b0 : y=y1;
        1'b1 : y=y2;
    endcase
endmodule
```

← 'always' block is used to describe 2:1 MUX at the input side. Select line is 'sel[0]' and output lines from input multiplexers are 'y1' and 'y2'.

← 'always' block is used to describe 2:1 MUX at the output side. 'sel[1]' selects the 'y1' or 'y2' output from input multiplexers

**Example 3.8** Synthesizable RTL for 4:1 MUX

## 3.2 Decoders

Decoder has ‘ $n$ ’ select lines or input lines and ‘ $m$ ’ output lines and used to generate either active high or active low output. The relation between select lines and output lines is given by  $m = 2^n$ . Depending on the logic status on ‘ $n$ ’ input lines at a time one of the output line goes high or low. Figure 3.6 represents 3:8 decoder; as shown in Fig. 3.6  $X_2$ ,  $X_1$ , and  $X_0$  are select inputs and from  $Y_0$  to  $Y_7$  are active high output lines.

The truth table of 3–8 decoder is shown in Table 3.3. For the active high output at a time one of the output line is active high.

Figure 3.7 is a symbolical representation of 3:8 decoder with active high enable input ‘en’. The truth table described above holds good for the decoder with active high enable ‘en=1’. When ‘en=0’ decoder is disabled and output ‘Y=8’b0000\_0000’. The synthesizable RTL is shown in the Example 3.9.

### 3.2.1 1 Line to 2 Decoder Using “case”

The 1 line to 2 or (1:2) decoder has one select input “Sel” and two output lines “Out\_Y0” and “Out\_Y1”. The truth table and equivalent representation is shown in Table 3.4 and Fig. 3.8, respectively.

The Verilog RTL is shown in the Example 3.10 and the equivalent synthesized result in Fig. 3.9.

### 3.2.2 1 Line to 2 Decoder with Enable Using “case”

The 1 line to 2 or (1:2) decoder has one select input “Sel”, enable input “En” and two output lines “Out\_Y0” and “Out\_Y1”. The truth table and equivalent representation are shown in Table 3.5.

The Verilog RTL is shown in the Example 3.11 and the equivalent synthesized result in Fig. 3.10.

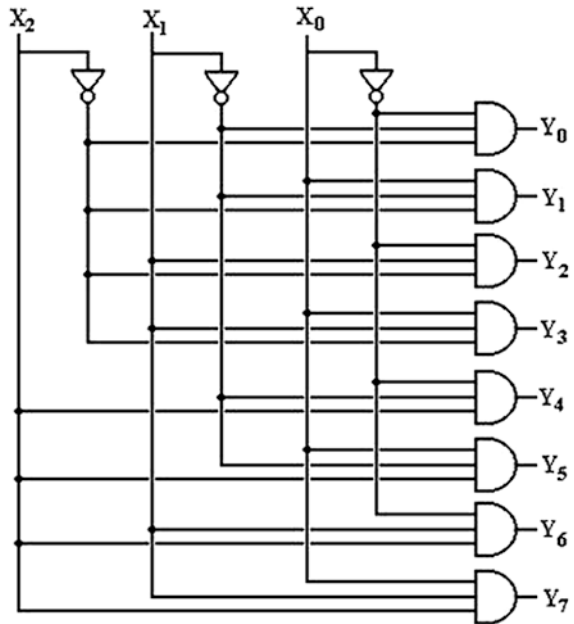
### 3.2.3 2 Line to 4 Decoder with Enable Using “case”

The 2 line to 4 or (2:4) decoder has two select inputs “Sel [1], Sel [0],” enable input “En” and four output lines “Out\_Y[3], Out\_Y[2], Out\_Y[1], and Out\_Y[0]”. The truth table and equivalent representation is shown in Table 3.6.

The Synthesizable Verilog RTL is described in the Example 3.12 and the equivalent hardware inferred is shown in Fig. 3.11.



**Fig. 3.6** Gate level representation of 3:8 decoder

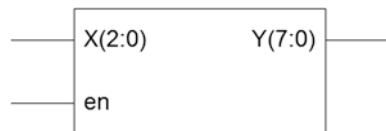


**Table 3.3** Truth table of 3:8 decoder

$X_2$	$X_1$	$X_0$	$Y_7$	$Y_6$	$Y_5$	$Y_4$	$Y_3$	$Y_2$	$Y_1$	$Y_0$
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

*Note* In the practical applications, decoders are used to select one of the memory or input–output device at a time. To enable the expansion of decoder, decoder always has either active high enable or active low enable input

**Fig. 3.7** Block representation of 3:8 decoder



```
//Verilog RTL for 3 to 8 decoder
module decoder_3to8 (X,en,Y);
input [2:0] X;
input en;
output [7:0] Y;
reg [7:0] Y;
// Functionality of design
always @ (X or en)
begin
    if (en)
        case(X)
            3'b000 : Y= 8'b0000_0001;
            3'b001 : Y= 8'b0000_0010;
            3'b010 : Y= 8'b0000_0100;
            3'b011 : Y= 8'b0000_1000;
            3'b100 : Y= 8'b0001_0000;
            3'b101 : Y= 8'b0010_0000;
            3'b110 : Y= 8'b0100_0000;
            3'b111 : Y= 8'b1000_0000;
        endcase
    else
        Y=8'b0000_0000;
end
endmodule
```

The decoder is enable for 'en=1' and generates one of the output as active high. The 'case' statement is used to describe the functionality of the decoder. Decoder generates parallel output.

In the RTL Verilog code 'if-else' is used to specify the select condition depending on the status of 'en'.



**Example 3.9** Verilog RTL for 3:8 Decoder

**Table 3.4** Truth table for 1:2 decoder

Sel	Out_Y1	Out_Y0
0	0	1
1	1	0

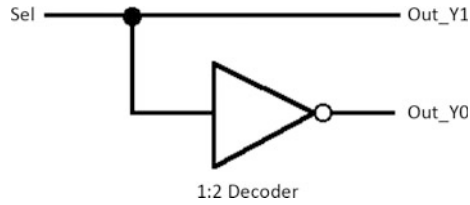


Fig. 3.8 1 line to 2 line decoder

```
//Verilog RTL for 1 Line to 2 Line decoder
module one_two_decoder(Sel, Out_Y1, Out_Y0);
input Sel;
output Out_Y1;
output Out_Y0;
reg Out_Y1;
reg Out_Y0;
always @ (Sel)
begin
    case (Sel)
        1'b0 : {Out_Y1, Out_Y0} = 2'b01;
        1'b1 : {Out_Y1, Out_Y0} = 2'b10;
    endcase
end
endmodule
```

The decoder generates active high output 'Out\_Y1, Out\_Y0' depending on the select input 'sel'.  
Sel=1 generates output as '10'.  
Sel=0 generates output as '01'

Example 3.10 Verilog RTL for 1:2 decoder

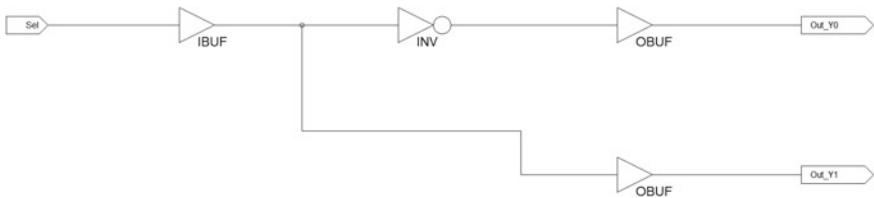


Fig. 3.9 Synthesized 1:2 decoder

**Table 3.5** Truth table for 1:2 decoder with enable

En	Sel	Out_Y1	Out_Y0
1	0	0	1
1	1	1	0
0	X	0	0

```
//Verilog RTL for 1 Line to 2 Line decoder with active high enable input
module one_two_decoder_with_enable ( Sel, En, Out_Y1, Out_Y0);
input Sel;
input En;
output Out_Y1;
output Out_Y0;
reg Out_Y1;
reg Out_Y0;
always @ (Sel or En)
begin
    if (En)
        case (Sel)
            1'b0 : {Out_Y1, Out_Y0} = 2'b01;
            1'b1 : {Out_Y1, Out_Y0} = 2'b10;
        endcase
    else
        {Out_Y1, Out_Y0} = 2'b00;
    end
end
endmodule
```

The decoder generates active high output 'Out\_Y1, Out\_Y0' depending on the select input 'sel'. For enable input 'En=1'

Sel=1 generates output as '10'.

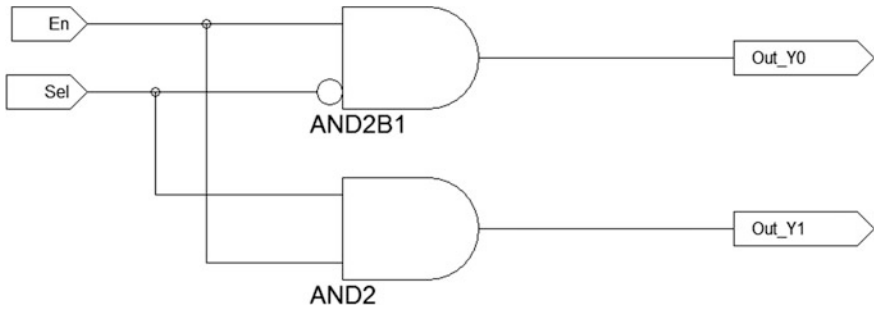
Sel=0 generates output as '01'

For 'en=0' Output is '00'

**Example 3.11** Verilog RTL for 1:2 decoder with enable input

### 3.2.4 2 Line to 4 Decoder with Active Low Enable Using 'case'

The 2 line to 4 or (2:4) decoder has two select inputs "Sel [1], Sel [0]," active low enable input "En\_bar" and four active low output lines "Out\_Y[3], Out\_Y[2],



**Fig. 3.10** Synthesized logic for 1:2 decoder with active high enable

**Table 3.6** Truth table for 2:4 decoder

En	Sel[1]	Sel[0]	Out_Y[3]	Out_Y[2]	Out_Y[1]	Out_Y[0]
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0
0	X	X	0	0	0	0

Out\_Y[1], and Out\_Y[0]”. The truth table and equivalent representation are shown in Table 3.7.

The Synthesizable Verilog RTL is described in the Example 3.13 and the equivalent hardware inferred is shown in Fig. 3.12.

Both the RTL Verilog codes described in the Example 3.13 infer the same hardware and are shown in Fig. 3.12.

### 3.2.5 4 Line to 16 Decoder Using 2:4 Decoder

The 4 line to 16 or (4:16) decoder has four select inputs “Sel [3]: Sel [0],” active low enable input “En\_bar” and designed by using 2:4 decoder which has four active low output lines “Out\_Y[3], Out\_Y[2], Out\_Y[1], and Out\_Y[0]”. The equivalent representation is shown in Fig. 3.13 (Example 3.14).

```
//Verilog RTL for 2 Line to 4 Line decoder with active high enable input
```

```
module Two_to_Four_decoder(Sel,En, Out_Y);
```

```
input [1:0] Sel;
```

```
input En;
```

```
output [3:0] Out_Y;
```

```
reg [3:0] Out_Y;
```

```
always @ (Sel or En)
```

```
begin
```

```
  if (En)
```

```
    case (Sel)
```

```
      2'b00 : Out_Y = 4'b0001;
```

```
      2'b01 : Out_Y = 4'b0010;
```

```
      2'b10 : Out_Y = 4'b0100;
```

```
      2'b11 : Out_Y = 4'b1000;
```

```
    endcase
```

```
  else
```

```
    Out_Y = 4'b0000;
```

```
end
```

```
endmodule
```



The decoder generates active high output ' Out\_Y[3]:Out\_Y[0] ' depending on the select input 'sel[1:0]'. For enable input 'En=1' For 'en=0' Output is '0000'

**Example 3.12** Synthesizable Verilog RTL for 2:4 decoder

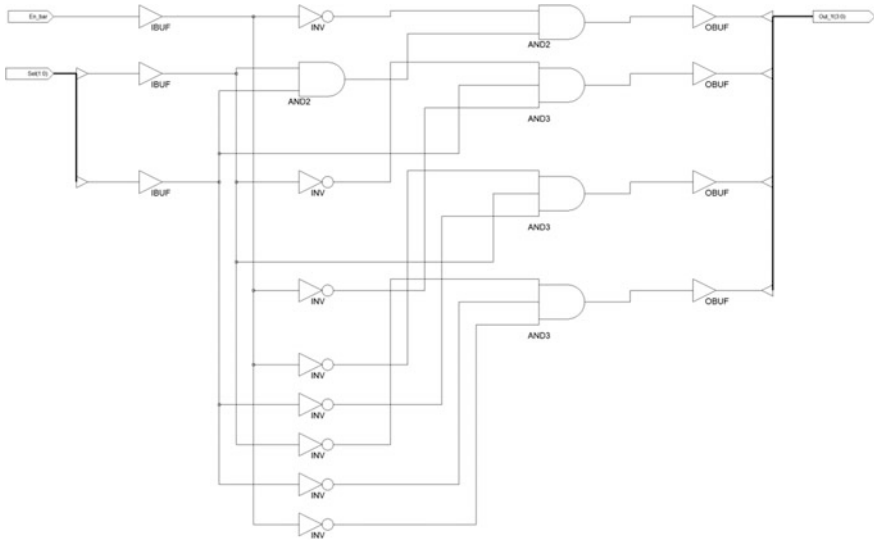


Fig. 3.11 2:4 Decoder with active high enable input

Table 3.7 Truth table for 2:4 decoder with active low enable and active low output

En_bar	Sel[1]	Sel[0]	Out_Y[3]	Out_Y[2]	Out_Y[1]	Out_Y[0]
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1
1	X	X	1	1	1	1

```
//Verilog RTL for 2 Line to 4 Line decoder with active low enable input and active low output lines
module Two_to_Four_decoder(Sel,En_bar, Out_Y);
input [1:0] Sel;
input En_bar;
output [3:0] Out_Y;
reg [3:0] Out_Y;
always @ (Sel or En_bar)
begin
    if (~En_bar)
        case (Sel)
            2'b00 : Out_Y = 4'b1110;
            2'b01 : Out_Y = 4'b1101;
            2'b10 : Out_Y = 4'b1011;
            2'b11 : Out_Y = 4'b0111;
        endcase
    else
        Out_Y = 4'b1111;
end
endmodule
//Verilog RTL for 2 Line to 4 Line decoder with active low enable input and for active low output lines
module Two_to_Four_decoder(Sel,En_bar, Out_Y);
input [1:0] Sel;
input En_bar;
output [3:0] Out_Y;
assign Out_Y[3] = (~En_bar) && (~Sel[1]) && (~Sel[0]);
assign Out_Y[2] = (~En_bar) && (~Sel[1]) && (Sel[0]);
assign Out_Y[1] = (~En_bar) && (Sel[1]) && (~Sel[0]);
assign Out_Y[0] = (~En_bar) && (Sel[1]) && (Sel[0]);
endmodule
```



The decoder generates active low outputs ' Out\_Y[3]:Out\_Y[0] ' depending on the select input 'sel[1:0]'. For enable input 'En\_bar=0' For 'En\_bar=1' Output is '1111'



The decoder generates active low outputs ' Out\_Y[3]:Out\_Y[0] ' depending on the select input 'sel[1:0]'. For enable input 'En\_bar=0' For 'En\_bar=1' Output is '1111'

**Example 3.13** Synthesizable RTL for 2:4 decoder



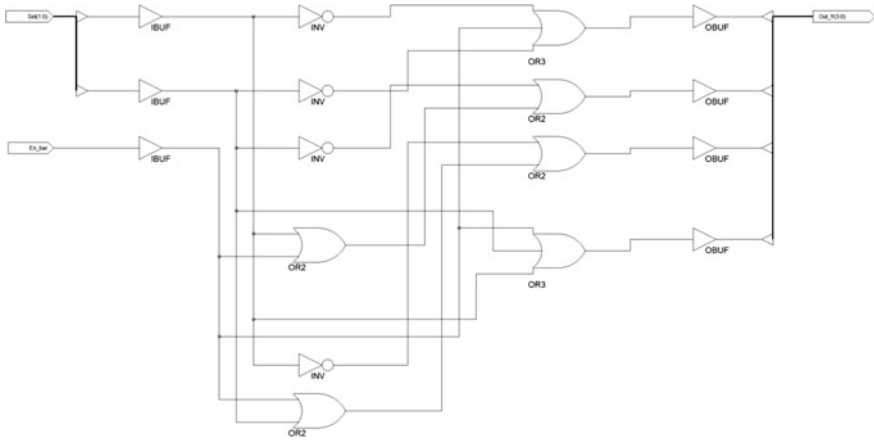


Fig. 3.12 Synthesizable hardware for 2:4 decoder

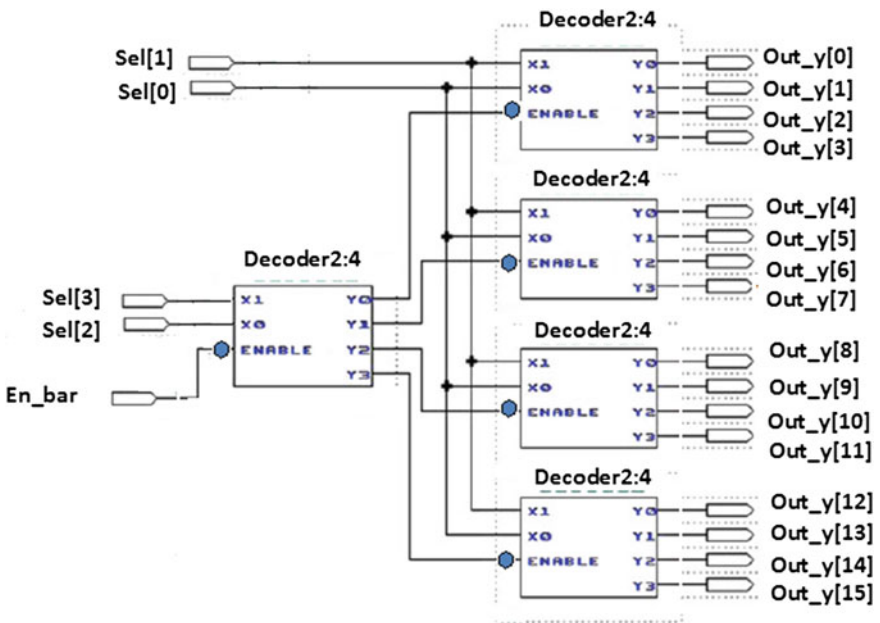


Fig. 3.13 Synthesized logic for 4:16 decoder using 2:4 decoders

```
//Verilog RTL for 4 to 16 decoder with active low enable and active low output
module four_to_sixteen_decoder(Sel,En_bar, Out_Y);
input [3:0] Sel;
input En_bar;
output [15:0] Out_Y;
reg [15:0] Out_Y;
reg [3:0] Out_Y_temp;
always @ (Sel[3:2] or En_bar)
begin
    if (~En_bar)
        case (Sel[3:2])
            2'b00 : Out_Y_temp = 4'b1110;
            2'b01 : Out_Y_temp = 4'b1101;
            2'b10 : Out_Y_temp = 4'b1011;
            2'b11 : Out_Y_temp = 4'b0111;
        endcase
    else
        Out_Y_temp= 4'b1111;
end
always @ (Sel[1:0] or Out_Y_temp[0])
begin
    if (~Out_Y_temp[0])
        case (Sel[1:0])
            2'b00 : Out_Y[3:0] = 4'b1110;
            2'b01 : Out_Y[3:0] = 4'b1101;
            2'b10 : Out_Y[3:0]= 4'b1011;
            2'b11 : Out_Y[3:0]= 4'b0111;
        endcase
    else
        Out_Y[3:0] = 4'b1111;
end
```

The decoder at input side is enabled when 'En\_bar=0' and generates the active low Output 'Out\_Y\_temp[3:0] '. This decoder is used to generate the enable signal for the output decoders.



The 2:4 decoder at the output side is enabled when 'Out\_Y\_temp[0]=0' and generates active low outputs 'Out\_Y[3:0]':



**Example 3.14** Synthesizable Verilog RTL for 4:16 decoder

```
always @ (Sel[1:0] or Out_Y_temp[1])
begin
    if (~Out_Y_temp[1])
        case (Sel[1:0])
            2'b00 : Out_Y[7:4] = 4'b1110;
            2'b01 : Out_Y[7:4] = 4'b1101;
            2'b10 : Out_Y[7:4] = 4'b1011;
            2'b11 : Out_Y[7:4] = 4'b0111;
        endcase
    else
        Out_Y[7:4] = 4'b1111;
end

always @ (Sel[1:0] or Out_Y_temp[2])
begin
    if (~Out_Y_temp[2])
        case (Sel[1:0])
            2'b00 : Out_Y[11:8] = 4'b1110;
            2'b01 : Out_Y[11:8] = 4'b1101;
            2'b10 : Out_Y[11:8] = 4'b1011;
            2'b11 : Out_Y[11:8] = 4'b0111;
        endcase
    else
        Out_Y[11:8] = 4'b1111;
end

always @ (Sel[1:0] or Out_Y_temp[3])
begin
    if (~Out_Y_temp[3])
        case (Sel[1:0])
            2'b00 : Out_Y[15:12] = 4'b1110;
            2'b01 : Out_Y[15:12] = 4'b1101;
            2'b10 : Out_Y[15:12] = 4'b1011;
            2'b11 : Out_Y[15:12] = 4'b0111;
        endcase
    else
        Out_Y[15:12] = 4'b1111;
end
endmodule
```

The 2:4 decoder at output side is enabled when 'Out\_Y\_temp[1]=0' and generates the active low Output 'Out\_Y [7:4]'.

The 2:4 decoder at output side is enabled when 'Out\_Y\_temp[2]=0' and generates the active low Output 'Out\_Y [11:8]'.

The 2:4 decoder at output side is enabled when 'Out\_Y\_temp[3]=0' and generates the active low Output 'Out\_Y [15:12]'.

Example 3.14 (continued)

### 3.3 Encoders

Function of an encoder is reverse of the decoder. Encoder has ‘ $n$ ’ input lines and ‘ $m$ ’ output lines and the relation between input lines and output lines is given by  $n = 2^m$ . For example consider 4:2 encoder. Number of input lines are  $n = 4$  and output lines  $m = 2$ . The block diagram of 4:2 encoder is shown in Fig. 3.14 with the equivalent gate level representation for 4:2 encoder and the truth table is described in Table 3.8.

The Verilog RTL description for 4:2 encoder is described in the Example 3.15. The Verilog RTL infers the similar hardware as shown in Fig. 3.14. As described in the Example 3.15 the output is ‘00’ when none of the input is active high and even when  $In\_A(0)$  is active high. So to indicate the difference the status flag need to be incorporated in the design. For  $In\_A(0) = 1$  output ‘00’ and status flag is one. For the default condition the status flag should be zero.

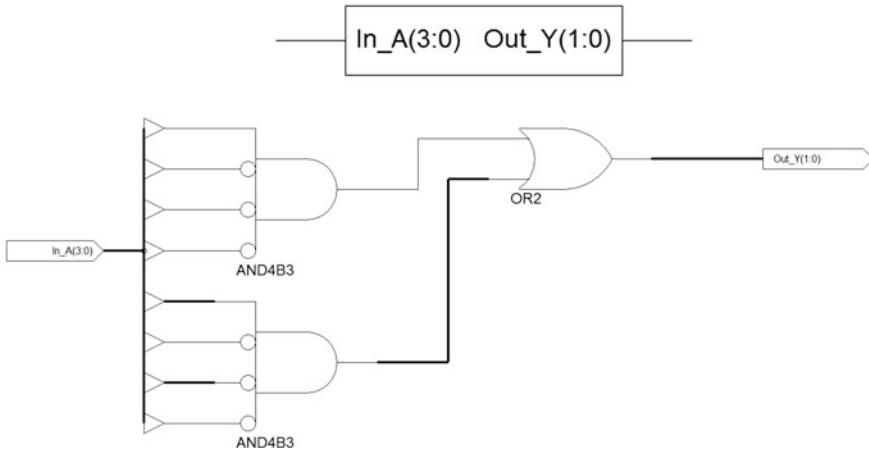


Fig. 3.14 The 4:2 encoder

Table 3.8 Truth table for 4:2 encoder

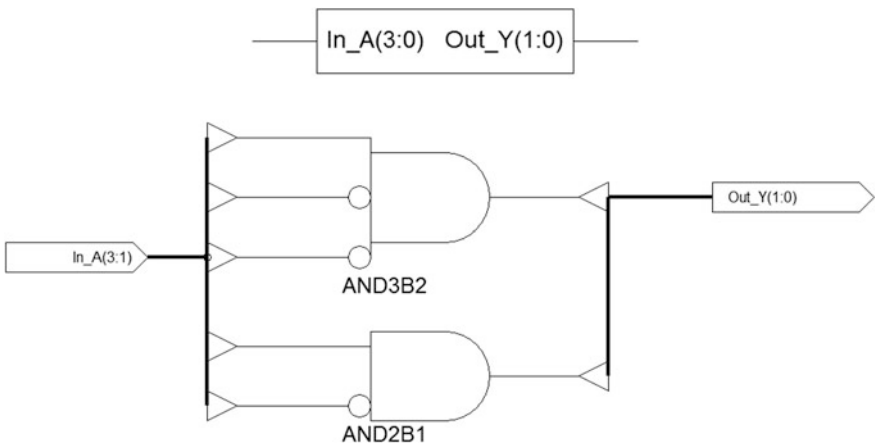
In[3]	In[2]	In[1]	In[0]	Out_Y[1]	Out_Y[0]
1	0	0	0	1	1
0	1	0	0	1	0
0	0	1	0	0	1
0	0	0	1	0	0

```
// Verilog RTL code for 4 to 2 encoder
module encoder_4to2 (In_A,Out_Y);
input [3:0] In_A;
output [1:0] Out_Y;
reg [1:0] Out_Y;
// Functionality of design
always @ (In_A)
begin
    case(In_A)
        4'b0001 : Out_Y= 2'b00;
        4'b0010 : Out_Y= 2'b01;
        4'b0100 : Out_Y= 2'b10;
        4'b1000 : Out_Y= 2'b11;
        default : Out_Y= 2'b00;
    endcase
end
endmodule
```

Encoder 4:2 is described using 'case' statement and the default value of the output is '00' if none of the condition specified using 'In\_A' is met.



**Example 3.15** Synthesizable Verilog RTL for 4:2 encoder



**Fig. 3.15** The 4:2 priority encoder

**Table 3.9** Truth table for priority 4:2 encoder

In_A[3]	In_A[2]	In_A[1]	In_A[0]	Out_Y[1]	Out_Y[0]
1	X	X	X	1	1
0	1	X	X	1	0
0	0	1	X	0	1
0	0	0	1	0	0

```
// Verilog RTL code for 4 to 2 priority encoder
```

```
module priority_encoder_4to2 (In_A,Out_Y);
```

```
input [3:0] In_A;
```

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

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

```
// Functionality of design
```

```
always @ (In_A)
```

```
begin
```

```
    if (In_A[3]==1'b1)
```

```
        Out_Y = 2'b00;
```

```
    else if ( In_A[2]==1'b1 )
```

```
        Out_Y = 2'b10;
```

```
    else if ( In_A[1]==1'b1 )
```

```
        Out_Y = 2'b01;
```

```
    else
```

```
        Out_Y = 2'b00;
```

```
end
```

```
endmodule
```

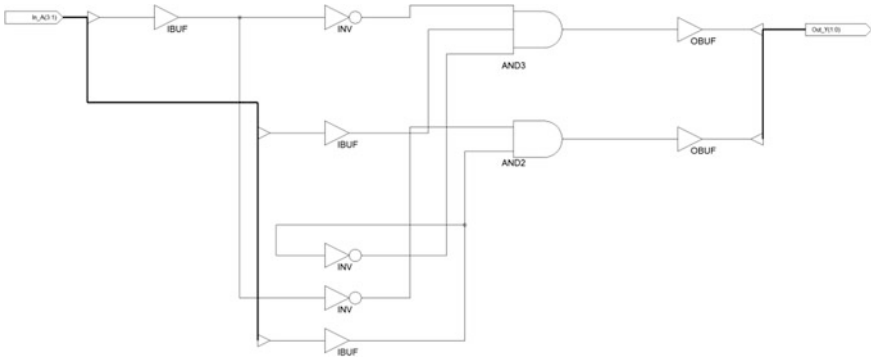


Priority Encoder 4:2 is described using 'if-else' statement and it generates the priority logic. As described 'In\_A[3]' has highest priority and 'In\_A[0]' has lowest priority.

**Example 3.16** Synthesizable Verilog RTL for 4:2 priority encoder

### 3.3.1 Priority Encoders

Priority encoders are used in the practical applications and has 'n' input lines and 'm' output lines and the relation between input lines and output lines is given by  $n = 2^m$ . For example consider 4:2 priority encoder. Number of input lines are  $n = 4$  and output lines  $m = 2$ . The block diagram of 4:2 priority encoder is shown in Fig. 3.15 with the equivalent gate level representation for 4:2 priority encoder and



**Fig. 3.16** Synthesized 4:2 priority encoder logic. *Note* In the practical applications, encoders are used to design the control logic. As “case” generates the parallel logic and “if-else” generates the priority logic; “case” is used to describe behavior of encoder. “if-else” is used to describe behavior of priority encoder. Priority encoders are used to sense the level sensitive interrupts

the truth table is described in Table 3.9. The input In\_A[3] has highest priority and the input In\_A[0] has lowest priority. Where ‘X’ indicates the do not care.

The Verilog RTL description for 4:2 priority encoder is described in the Example 3.16. The Verilog RTL infers the hardware as shown in Fig. 3.16.

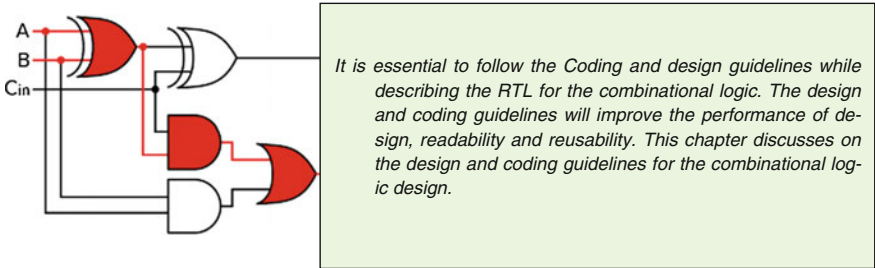
### 3.4 Summary

As discussed in this chapter the combinational logic RTL using Verilog can be efficiently written using the Verilog constructs and following are key points to summarize:

1. “assign” is used to infer the 2:1 MUX. MUX is treated as universal logic cell.
2. “if-else” generates the 2:1 MUX and “nested if” generates the priority logic.
3. “case-endcase” is used to model the parallel logic and used inside the procedural block.
4. “default” condition in the “case-endcase” is used to describe the nonspecified conditions covered in the case.
5. Synthesis tool ignores the sensitivity list specified in the procedural blocks.
6. Decoders are used to select one of the memory or input–output device at a time.
7. Priority encoders are used in the design of interrupt control logic and logic can be described by using nested “if-else”.

# Chapter 4

## Combinational Design Guidelines



**Abstract** This chapter describes about the design guidelines for the combinational logic designs. In the practical ASIC designs, these guidelines are used to improve the readability, performance of the design. The key practical guidelines discussed are use of ‘if-else’ and ‘case’ constructs and the practical scenarios, how to infer the parallel and priority logic. The detailed practical use of resource sharing and use of blocking assignments to describe the combinational logic design is explained in detail. The chapter key highlight is the description of the stratified event queuing and logical partitioning. This chapter also describes the scenarios of missing else, default in the sequential statements and combinational looping in the design. All the guidelines in this chapter are covered with the meaningful practical examples and the synthesized logic is explained for better understanding.

**Keywords** Stratified event queue · Logical partitioning · Active · Inactive · NBA · Monitor · Postponed · Delay assignments · Logical equality · Case equality · Logical inequality · Case inequality · Blocking · Non-blocking · if-else · case · Default · Sensitivity · Looping · Race conditions · Oscillatory behavior · Multiple driver

The design and coding guidelines are used to improve the design performance, readability and the reusability for the design. The combinational design where the output is function of the present inputs should be described in such a way that the design should have least propagation delay time and the least area.



It is always recommended to follow certain coding guidelines while describing the design using Verilog RTL. This chapter is more focused on the key design and coding guidelines used in the industries to describe an efficient combinational logic.

## 4.1 Use of Blocking Assignments and Event Queue

Verilog supports the two kinds of the assignments in the procedural blocks. These assignments are named as blocking ( $=$ ) and non-blocking ( $<=>$ ) assignments. It is always recommended to use the blocking assignments while describing the combinational logic design. The reason being very simple to understand but the essence is to understand the fundamental behind it as an engineer.

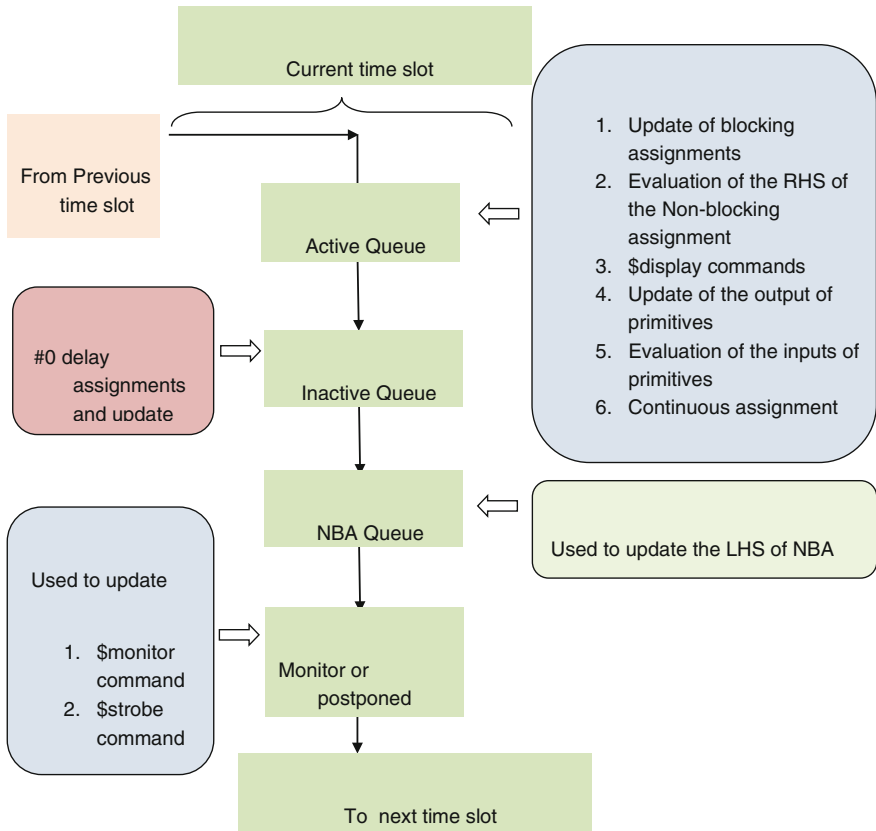
To have the understanding of the blocking assignments, let us understand the concept of stratified event queue. According to IEEE 1364-2005 Verilog standard, the stratified event queue is divided into four major regions. These regions are named as: Active, Inactive, NBA, and Monitor.

But the major question is why to have the understanding of the stratified event queue and what exactly is the application of it. As the name itself indicates that the stratified event queue is used to evaluate the expressions and update the results. Figure 4.1 describes the stratified event queue according to the Verilog IEEE 1364-2005 standard.

As shown in Fig. 4.1 the Verilog Stratified Event Queue has four main regions and are explained below

- i. *Active Queue* Most of the Verilog events are scheduled in the active event queue. These events can be scheduled in any order and evaluated or updated in any order. The active queue is used to update the blocking assignments, continuous assignments, evaluation of RHS of the non-blocking assignments (LHS of NBA is not updated in the active queue), \$display commands and the updating the primitives.
- ii. *Inactive Queue* The #0delay assignments are updated in the inactive queue. Using #0 delays in the Verilog is not good practice and it unnecessarily complicates the event scheduling and ordering. Most of the times, the designer uses the #0 delay assignments to fool the simulator to avoid the race around conditions.
- iii. *NBA Queue* The LHS of the non-blocking assignments updates in this queue.
- iv. *Monitor Queue* it is used to evaluate and update the \$monitor and \$strobe commands. The updates of all the variables are during the current simulation time.

As discussed above, the blocking assignments execute sequentially inside procedural block. Blocking assignment blocks all the trailing statements in the procedural block while executing the current statement. The execution of the blocking



**Fig. 4.1** Verilog stratified event queue

assignment is always considered as one-step process. In an active event queue, the RHS of blocking assignment is evaluated and during the same time stamp the LHS of blocking assignment is updated. Consider an Example 4.1 for the blocking assignments.

In the subsequent section we will discuss on the design and coding guidelines for combinational logic and we will continue to use the blocking assignments.

## 4.2 Incomplete Sensitivity List

It is recommended to incorporate all the required signals and inputs in the sensitivity list of combinational design procedural block. Consider the Example 4.2 to describe the functionality of two input NAND logic.

```
// Verilog RTL code for blocking assignments
```

```
reg a_reg, b_reg;
```

```
// Functionality of design
```

```
always @ (a_reg or b_reg)
```

```
begin
```

```
    a_reg=b_reg;
```

```
    b_reg=a_reg;
```

```
end
```

In the Verilog RTL code if initial value at the current time slot for a\_reg=9 and b\_reg=8 then at the end of execution of the always block the result is a\_reg=8 and b\_reg=8.

Due to use of the blocking assignments the statements inside always block executes sequentially.

**Example 4.1** Blocking assignments update in the procedural block. *Note* The major issue with the blocking assignments is during the use of the same variable on the RHS side in one procedural block and on LHS side in another procedural block. If both the procedural blocks are scheduled in the same simulation time or on the same clock edge, it generates the race condition in the design. This will be discussed in the subsequent chapters

In the Example 4.2, the synthesis tool ignores the sensitivity list and generates the two input NAND gate as synthesizable output but the simulator ignores the changes in the input 'b\_in' and generates the output waveform. This leads to simulation and synthesis mismatch. The simulation result is shown in Fig. 4.2.

### 4.3 Continuous Versus Procedural Assignments

Continuous assignments: Continuous assignments are used to assign the value to the net. They are used to describe the combinational logic functionality. These assignments are updated in the active event queue and the net values are updated upon evaluation of the right-hand side expression. The port or output is declared as 'wire' while using the continuous assignment statement.

```
assign y_out= sel_in ? a_in : b_in;
```

```
// Verilog RTL code for incomplete sensitivity list

module logic_design(a_in,b_in,y_out);

input wire a_in;

input wire b_in;

output reg y_out;

// Functionality of design

always @ (a_in)

begin
    if (a_in==1'b1 && b_in==1'b1)

        y_out = 1'b0;

    else

        y_out =1'b1;

end

endmodule
```

In the sensitivity list the a\_in is specified but b\_in input is missing and the simulator and synthesis Tool will flash the warning "Incomplete Sensitivity list"



**Example 4.2** Incomplete sensitivity list

Procedural assignments: Procedural assignments are used to assign value to the variable reg. These are used to describe both the combinational and sequential logic behavior. The output assigned to reg is held until the next assignment is executed. These assignments are used in the procedural blocks always, initial and inside the task and functions.

```

always@(posedge clk) // Sequential design description

begin

q_out<= data_in;

end

always@(a,b) // Combinational design description

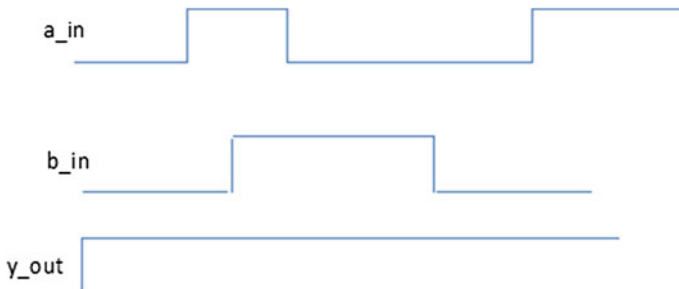
begin

y_out = sel_in ? a_in : b_in;

end

```

In the procedural block, if the blocking ( $=$ ) assignments are used, they are updated in the active event queue. All the non-blocking assignments ( $<=$ ) are evaluated in the active event queue but updated in the non-blocking event queue.



**Fig. 4.2** Incomplete sensitivity list waveform. *Note* To avoid the simulation and synthesis mismatch it is recommended to use the procedural block: `always@(*)`. According to IEEE 1364-2001 standard the `*` in the sensitivity list will include all the inputs and required signals

## 4.4 Combinational Loops in Design

The unintentional combinational loops in the design are very critical to debug and fix during the implementation phase and generates an oscillatory behavior. The Example 4.3 describes the combinational loop in the design.

Figure 4.3 describes the synthesizable output for the combinational loop.

As discussed above, combinational loops in design are one of the dangerous and critical design errors. Combinational loop in the design occurs in the same signal are used or updated in the multiple procedural blocks. If the same signal is present on the right-hand side of expression and on the left hand side of expression, the design has combinational loop.

Combinational loops exhibit the oscillatory behavior and during updating, they can have race conditions. Consider the design scenario shown in Example 4.4.

In Example 4.4, both always blocks execute concurrently and due to that, while updating the  $b$  value the new value to  $a$  is assigned. This has race condition in the design. This design generates the oscillatory behavior due to events on  $a$ ,  $b$ .

```
// Verilog RTL code for unintentional combinational loop
```

```
module loop_combinational(a_in,b_in,y_out);
```

```
input wire a_in;
```

```
input wire b_in;
```

```
output reg y_out;
```

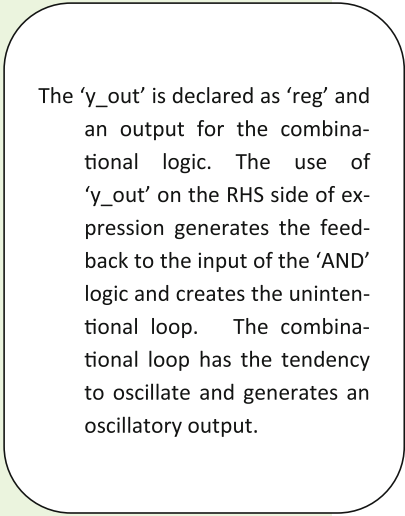
```
// Functionality of design
```

```
always @ (a_in, b_in)
```

```
begin
```

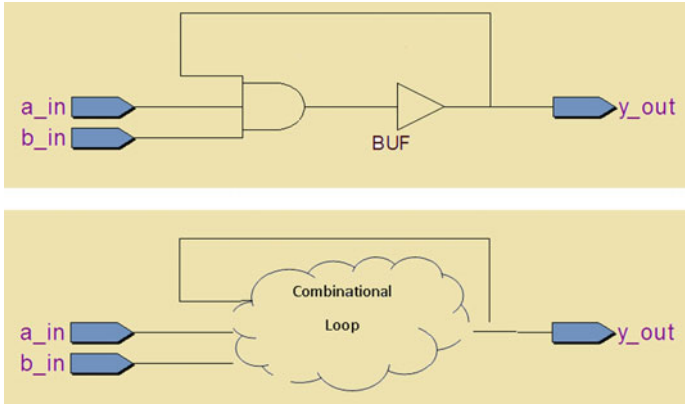
```
    y_out = a_in & b_in & y_out;
end
```

```
endmodule
```



The 'y\_out' is declared as 'reg' and an output for the combinational logic. The use of 'y\_out' on the RHS side of expression generates the feedback to the input of the 'AND' logic and creates the unintentional loop. The combinational loop has the tendency to oscillate and generates an oscillatory output.

**Example 4.3** Combinational loop in the design



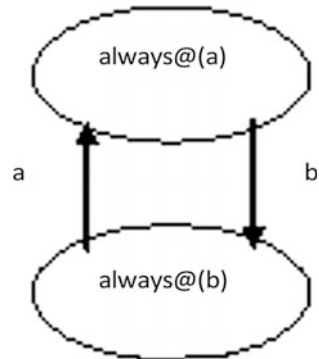
**Fig. 4.3** Combinational loop outcome. *Note* It is recommended that the design should not have any combinational loop. To avoid the combinational loop break the feedback path by using the sequential elements

**Example 4.4** Verilog RTL code with combinational loop

```
always@(a)
begin
    b=a;
end

always@(b)
begin
    a=b;
end
```

**Example 4.5** Oscillatory behavior due to combinational looping



**Example 4.6** Solution to break combinational loop

```
always@(posedge clk)
begin
    b<=a;
end

always@(posedge clk)
begin
    a<=b;
end
```

The oscillatory behavior can be understood from Example 4.5.

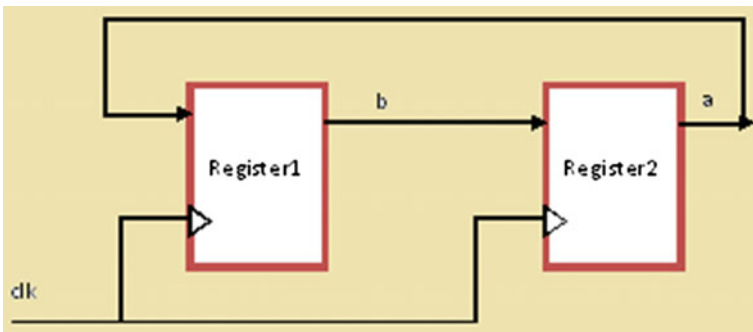
Combinational loops are not synthesizable and the synthesizer generates an error or warning for the combinational loop. Combinational looping can be potential hazard in the design and hence need to be avoided.

As shown in the above diagram the always block always@(a) is triggered on the event on a and generates an output b. Eventually, changes on b input are used to trigger another always block always@(b) and generates the output a. So this goes on and exhibits the oscillatory behavior or the race around conditions in the design.

The solution to overcome this problem is to use the register to avoid the dependency of signals to trigger multiple always block. Register can be inserted in the combinational loop to update the value.

To avoid combinational looping, do the following. Use the non-blocking assignments and use the register logic to break the combinational loops. The modification is shown in the Example 4.6.

In the Example 4.6, both always blocks are triggered on positive edge of clock and assigns the value to b, a respectively. Although both the procedural blocks are



**Fig. 4.4** Register logic to avoid oscillatory behavior



executed concurrently, the non-blocking assignments are queued in the NBA queue and hence generates the structure as shown in Fig. 4.4.

## 4.5 Unintentional Latches in the Design

It is recommended that the design should not have unintended latches as latch acts as transparent during active level and transfers the data to its output. The unintentional latches are not recommended in ASIC design as it causes the issues during the design testing or during DFT. Even during STA, the timing algorithm will be not able to understand whether to sample the data on positive edge of the clock or on negative edge of the clock. So, most of the time as real intention of the designer is not reflected in the hardware inference, the STA for such paths are difficult. This will be discussed in subsequent chapters.

Consider the functionality shown in the following Example 4.7.

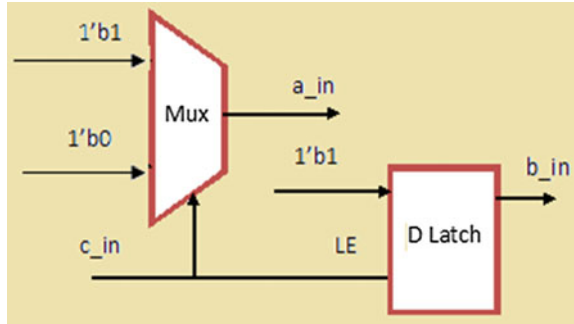
In the above code, as during the else clause the information about the update of b\_in is not given, it infers the latch and holds the previous value of b\_in. The diagrammatic representation is shown in Fig. 4.5. If-else statement infers multiplexer for a\_in assignment and for b\_in assignment it infers the positive level sensitive latch controlled by enable input c\_in.

As shown in Fig. 4.5, due to missing b\_in assignment in the else clause it generates the latch and holds the previous value assigned in the if clause. Latches

**Example 4.7** Verilog RTL with missing 'else' condition

```
always@(c_in)
begin
    if (c_in==1)
begin
a_in=1'b1;
b_in=1'b1;
end
else
begin
a_in=1'b0;
end
end
```

**Fig. 4.5** Synthesized logic with missing 'else' condition



are inferred due to the incomplete assignment in if-else or due to incomplete conditions covered in the case statements. It is recommended that designer should take care of this while writing a RTL code.

### 4.6 Use of Blocking Assignments

As discussed above, blocking assignments are denoted by (=) and used inside a procedural block to describe the functionality of combinational logic design. Readers are requested not to get confused with the (=) assignment used by using

```

module half_adder ( a_in, b_in, sum_out, carry_out);

input a_in;

input b_in;

output sum_out;

output carry_out;

assign sum_out = a_in ^ b_in;

assign carry_out = a_in & b_in;

endmodule
    
```

In the Verilog RTL code continuous assignment statement 'assign' is used so the expression uses (=) to assign the LHS expression value to RHS. But this assignment is continuous assignment. Both 'assign' statements will execute concurrently.

**Example 4.8** Continuous assignment Verilog RTL. *Note* It is recommended to use the full adder to perform the subtraction operation. Subtraction is performed using 2's complement addition. Multiple continuous assignment statements executes concurrently

```
module blocking_assignment ( a_in, y1_out, y2_out);  
  
input a_in;  
  
output reg y1_out;  
  
output reg y2_out;  
  
always@(a_in) ←  
  
begin  
  
y2_out=y1_out;  
  
y1_out=a_in;  
  
end  
  
endmodule
```

In the Verilog RTL code blocking assignments are used to describe the combinational behavior and the 'y2\_out' is assigned first and the 'a\_in' is assigned to 'y1\_out' last.

This results into simulation and synthesis mismatch. As during simulation it is essential to hold the previous value of 'a\_in'.

**Example 4.9** Blocking assignment inside procedural block

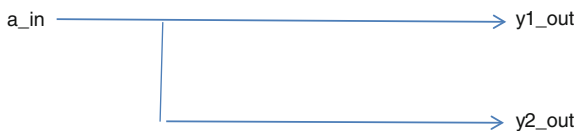
continuous assignment 'assign'. Example 4.8 uses the multiple assign constructs to describe the functionality of design.

Consider the scenario of use of blocking assignment in the procedural block. If the order of the blocking assignment is not proper, then there is chance for the simulation and synthesis mismatch.

Example 4.9 is shown and in the example, the issue in simulation and synthesis outcome is due to ordering of the blocking statements. Blocking assignment blocks the next immediate statement execution unless and until current statement is executed. Readers are encouraged to use only blocking assignments but care should be taken while using the statements to get the real intended results.

The synthesis result for the above example is shown in Fig. 4.6 and it generates two wires. But while simulating the 'y2\_out' is updated with the previous time stamp value 'a\_in'. So results in simulation and synthesis mismatch.

**Fig. 4.6** Synthesis result of blocking assignment



### 4.7 Use of If-Else Versus Case Statements

When all the case conditions covered in the ‘case-endcase’ the statement is said to be full-case statement. For combinational design, case statement should use all the blocking assignments.

The synthesis result for 4:1 MUX is shown in Fig. 4.7 and it infers parallel logic.

```

module mux_4to1(y_out,d_in,s_in);
input[3:0]d_in;
output y_out;
reg y_out;
input [1:0]s_in;
always @ (s_in or d_in)
begin
case (s_in)
2'b00 : y_out = d_in[0];
2'b01 : y_out = d_in[1];
2'b10 : y_out = d_in[2];
default : y_out = d_in[3];
endcase
end
endmodule
    
```

‘always’ procedural block describes the functionality of design. Full-case is used describe the functionality. Depending on ‘s\_in’ status one of the input d\_in[3:0] is connected to an output ‘y\_out’. Due to use of ‘case’ construct it generates parallel hardware.

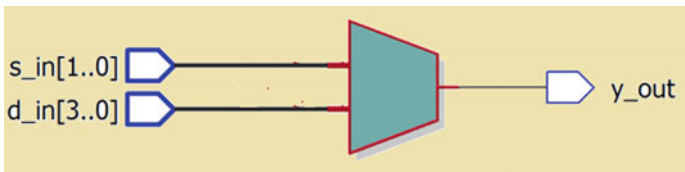


Fig. 4.7 Parallel logic inference for 4:1 MUX using ‘case’

```

module mux_4to1(y_out,d_in,s_in);
  input[3:0]d_in;
  output y_out;
  reg y_out;
  input [1:0]s_in;
  always @ (s_in or d_in)
  begin
    if(s_in == 2'b00)
      y_out = d_in[0];
    else if(s_in == 2'b01)
      y_out = d_in[1];
    else if(s_in == 2'b10)
      y_out = d_in[2];
    else
      y_out = d_in[3];
  end
endmodule

```

Nested 'if-else' structure generates priority logic. The functionality using 'if-else' is described for 4:1 MUX and in this 'd\_in[3]' has least priority and 'd\_in[0]' has the highest priority.

**Example 4.10** Verilog RTL for priority logic

## 4.8 MUX Nested or Priority Structure

If the functionality is described by using 'if-else' construct then the synthesis outcome results into priority logic. It is recommended to use 'if-else' construct to describe the priority logic. Example 4.10 describes the functionality of 4:1 MUX using nested if-else construct.

## 4.9 Decoder 2:4

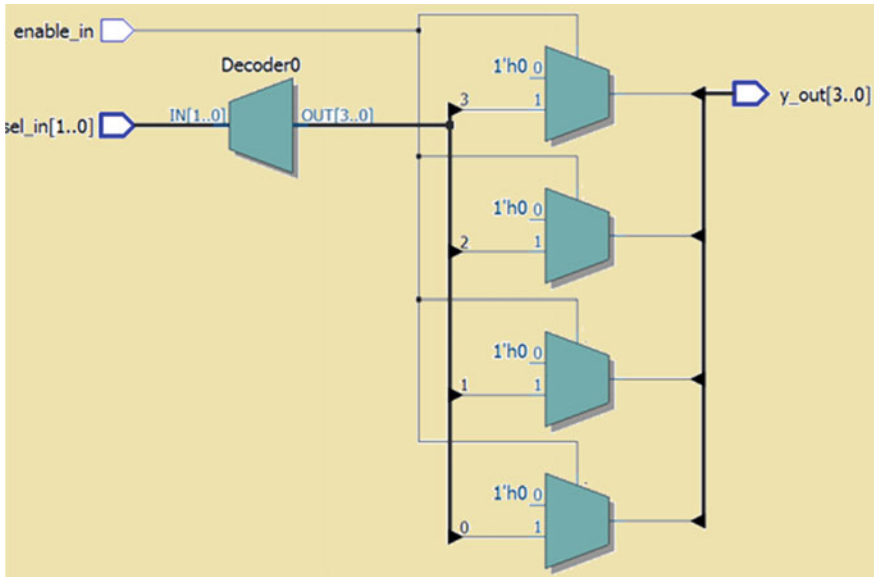
While describing the functionality of decoding logic, use the continuous assignment or 'case' construct. Both will generate the parallel logic. As discussed in Chap. 3, decoder has parallel select inputs and generates parallel outputs.

```

module decoder_2to4 ( sel_in, enable_in, y_out);
  input [1:0] sel_in;
  input enable_in;
  output wire [3:0] y_out;
  assign y_out = enable_in ? ( 1 << sel_in ) : 4'b0000;
endmodule

```

The decoder functionality is described by using 'assign' statement and it infers parallel logic.



**Fig. 4.8** Decoding logic using ‘assign’ or ‘case’

If the decoder is described using ‘case-endcase’ statement, then it also infers the parallel logic. The hardware description for decoder implementation using ‘assign’ and ‘case-endcase’ is shown in Fig. 4.8 (Example 4.11).

### 4.10 Encoder 4:2

To describe the encoder functionality, use the ‘if-else’ construct as priority definition can be defined. The functionality of 4:2 encoder is described using ‘if-else’ construct and it infers the priority logic. For Example 4.12, the synthesized outcome is shown in Fig. 4.9.

### 4.11 Missing ‘Default’ Clause in Case

If all conditions are not covered in the ‘case-endcase’ expression, then it infers the latches in the design. If all case conditions are not required in the design functionality, then it is recommended to use ‘default’ clause. If ‘default’ is missing, the synthesizer flashes warning for missing ‘case’ conditions and infers latches in the design.

```
//Decoder using 'case-endcase'
module decoder_2to4 ( sel_in, enable_in, y_out);

input [1:0] sel_in;
input enable_in;
output reg [3:0] y_out;

always @ (*)
begin
if (enable_in)
case (sel_in)
2'b00 : y_out = 4'b0001;
2'b01 : y_out = 4'b0010;
2'b10 : y_out = 4'b0100;
default : y_out = 4'b1000;
endcase
else
y_out = 4'b0000;
end
endmodule
```



In this 'always' procedural block is used with the 'case' construct to describe the functionality of the decoder.

For 'enable\_in=1' the decoder generates the valid output depending on the status of 'sel\_in'.

**Example 4.11** Decoding logic using 'case-endcase' construct

```
module encoder_4to2 ( data_in, y_out);
input [3:0] data_in;
output reg [1:0] y_out;
always @ (*)
begin
if (data_in[3])
y_out = 2'b00;
else if (data_in[2])
y_out = 2'b01;
else if (data_in[1])
y_out = 2'b10;
else
y_out = 2'b11;
end
endmodule
```



As described using 'always' block 'data\_in[3]' has highest priority over all the remaining inputs.

Input 'data\_in[0]' has the least priority.

**Example 4.12** Priority logic using 'if-else'

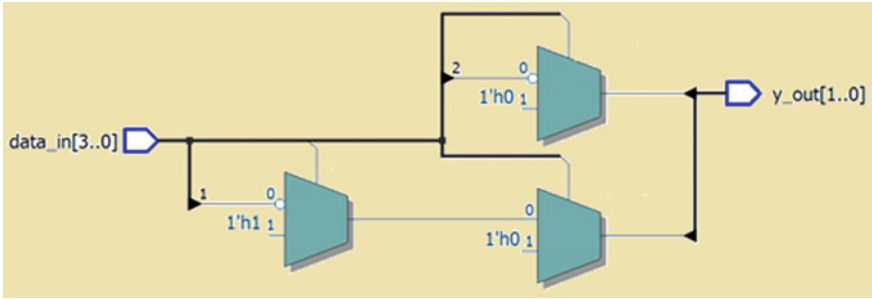


Fig. 4.9 Synthesized result of priority encoding using if-else'

```

module decoder_2to4 ( sel_in, enable_in, y_out);
    input [1:0] sel_in;
    input enable_in;
    output reg [3:0] y_out;

    always @ (*)
    begin
        if (enable_in)
            case (sel_in)
                2'b00 : y_out = 4'b0001;
                2'b01 : y_out = 4'b0010;
                2'b10 : y_out = 4'b0100;
            endcase
        else
            y_out = 4'b0000;
        end
    end
endmodule
    
```

In this 'case' expression is non-full as 2'b11 condition is not described. Even 'default' clause is not used to describe the functionality for the missing 'case' conditions.

This infers unintentional latches in the design

**Example 4.13** Missing default in case

The synthesis result for Example 4.13 is shown in Fig. 4.10.

### 4.12 If-Else with Else Missing

As shown in the example the 4:1 MUX functionality is described using nested 'if-else' but due to missing 'else' clause it infers 4:1 MUX with the unintentional latches. It is recommended to avoid the unintentional latches by incorporating the 'else' clause at the desired and required places in the RTL code.

For the Example 4.14, the similar hardware is generated as shown in Fig. 4.10.



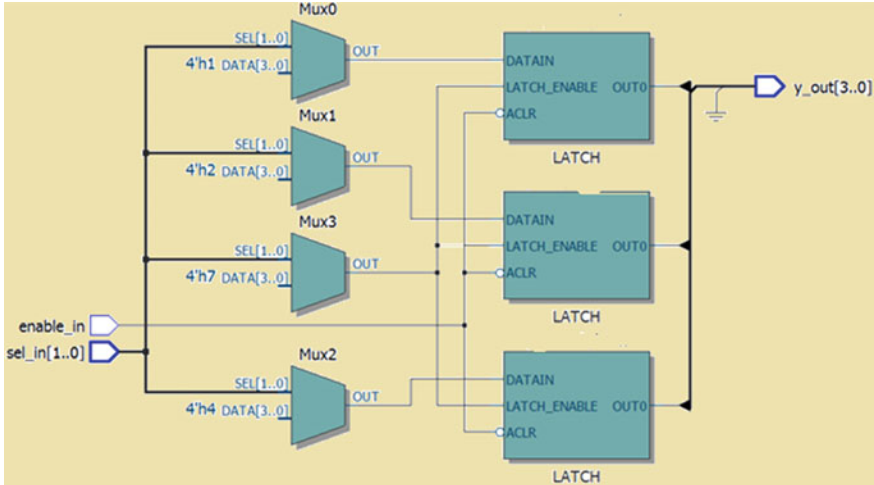


Fig. 4.10 Synthesized output for the missing default

```

module mux_4to1_else_mising (y_out,d_in,s_in);
input[3:0]d_in;
output y_out;
reg y_out;
input [1:0]s_in;
always @ (s_in or d_in)
begin
  if( s_in == 2'b00)
    y_out = d_in[0];
  else if( s_in == 2'b01)
    y_out = d_in[1];
  else if( s_in == 2'b10)
    y_out = d_in[2];
  else if( s_in == 2'b11)
    y_out = d_in[3];
end
endmodule

```

As else is missing in the nested-if the hardware inference generates combinational multiplexers with the latches.

The latches are inferred due to missing 'else' in the 'if-else' statement.

Example 4.14 Verilog RTL with missing 'else'

## 4.13 Logical Equality Versus Case Equality

Logical equality (==) and logical inequality (!=) operators are used in the synthesizable designs whereas case equality (===) and case inequality (!==) are not recommended in the synthesizable design.

### 4.13.1 Logical Equality and Logical Inequality Operators

1. Recommended to be used in the synthesizable design.
2. If any one of the operand has either 'x' or 'z' value, then the result is unknown ('x') and it results into logical comparison result as false.
3. The comparison outcome is non-deterministic if any one of the operand has 'x' or 'z' value.
4. Consider example of comparing 'a\_in' with 'b\_in'. In this, if either of the operand as 'x' or 'z' value, then the else clause will be executed and infers the logic specified in the else clause

```
always@(a_in, b_in)
begin
if (a_in==b_in)
y_out= a_in ^b_in;
else
y_out =a_in &b_in;
end
//For either of a_in, b_in has 'x' or 'z' value then the re-
sult is y_out= a_in & b_in;
```

### 4.13.2 Case Equality and Case Inequality Operators

1. Recommended to be used in the non synthesizable design.
2. If any one of the operand has either 'x' or 'z' value, then the result is known value and it results either true or false.
3. The comparison outcome is deterministic if any one of the operand has 'x' or 'z' value.

4. Consider example of comparing 'a\_in' with 'b\_in'. In this, if either of the operand as 'x' or 'z' value, then if a\_in is equal to b\_in the if clause will be executed and infers the logic specified in the if clause

```

always@(a_in, b_in)
begin
if (a_in===b_in)
y_out= a_in ^b_in;
else y_out =a_in &b_in;
end
//For either of a_in,b_in has 'x' or 'z' value then the re-
sult is y_out= a_in ^ b_in;

```

## 4.14 Arithmetic Resource Sharing

The Example 4.15, the **design** without resource sharing. The intended design functionality is to design the combinational logic shown in Table 4.1.

As shown in the synthesized logic in Fig. 4.11, it uses three full-adders and two multiplexers. The synthesized logic is inefficient as all the additions are performed simultaneously and multiplexer output is control signal dependent. So it is wastage of more power and inefficient as per as area utilization is concern.

### 4.14.1 With Resource Sharing

Resource sharing is one of the efficient techniques used in the ASIC design to share the common resources. As discussed in Example 4.15, the adders are generating the results simultaneously and waits for the control signal either 's1\_in' or 's2\_in' (Example 4.16, Fig. 4.12).

```

module logic_without_sharing( a_in, b_in, c_in, d_in, e_in, f_in,
    s1_in, s2_in, y_out, z_out);

input a_in;
input b_in;
input c_in;
input d_in;
input e_in;
input f_in;
input s1_in;
input s2_in;
output reg y_out;
output reg z_out;
always @ ( a_in, b_in, c_in, d_in, s1_in)
begin
    if ( s1_in )
        y_out = a_in + b_in;
    else
        y_out = c_in + d_in;
end

always @ ( a_in, b_in, e_in, f_in, s2_in)
begin
    if ( s2_in )
        z_out = e_in + f_in;
    else
        z_out = a_in + b_in;
end

endmodule
    
```



The combinational design is described by using two different 'always' blocks.

The addition operation is performed by using operator '+'. Due to 'if-else' construct it infers the 2:1 MUX.

**Example 4.15** Verilog RTL without using the concept of resource sharing

**Table 4.1** Functional table description

s1_in	y_out
1	a_in + b_in
0	c_in + d_in
s2_in	z_out
0	a_in + b_in
1	e_in + f_in

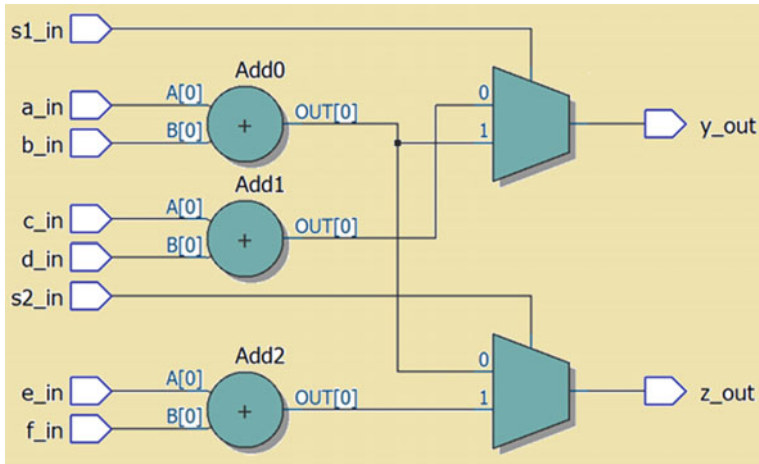


Fig. 4.11 Synthesized logic without resource sharing

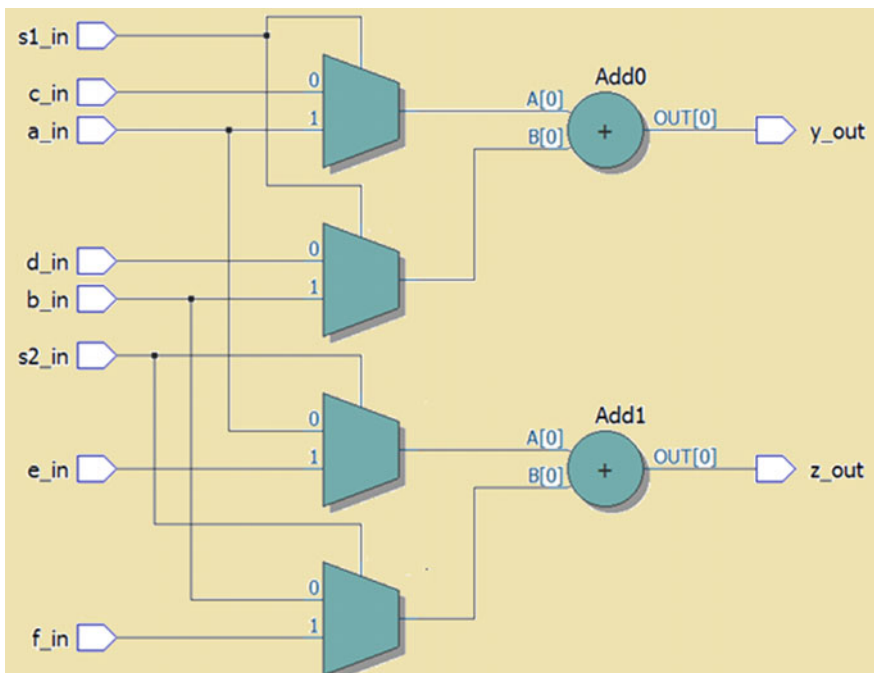


Fig. 4.12 Synthesized logic using common resources

```

module logic_with_sharing( a_in, b_in, c_in, d_in, e_in, f_in,
    s1_in, s2_in, y_out, z_out);

input a_in;
input b_in;
input c_in;
input d_in;
input e_in;
input f_in;
input s1_in;
input s2_in;
output reg y_out;
output reg z_out;
reg temp1,temp2,temp3,temp4;

always @ ( a_in, b_in, c_in, d_in, s1_in)
begin
    if ( s1_in )
        begin
            temp1 = a_in;
            temp2 = b_in;
        end
    else
        begin
            temp1 = c_in;
            temp2 = d_in;
        end
    end
end

always @ ( a_in, b_in, e_in, f_in, s2_in)
begin
    if ( s2_in )
        begin
            temp3 = e_in;
            temp4 = f_in;
        end
    else
        begin
            temp3 = a_in;
            temp4 = b_in;
        end
    end
end

always @ ( temp1, temp2, temp3, temp4 )
begin
    y_out = temp1 + temp2;
    z_out = temp3 + temp4;
end

endmodule
    
```

The combinational design is described by using two different 'always' blocks.

In this the functionality is described using multiple number of multiplexers at the input side to sample the required input.

Adders are used as common resources to perform addition depending on the inputs sampled by using multiple multiplexers.

**Example 4.16** Verilog RTL with resource sharing technique

**Example 4.17** Verilog RTL with multiple driver assignment

```
wire y_tmp;

assign y_tmp = a_in ^ b_in;

assign y_tmp = a_in & b_in;

//in this example y_tmp is assigned by using 'xor' and 'and' at
a time and hence multiple driver assignment error.
```

## 4.15 Multiple Driver Assignments

If same net (wire) is driven by multiple expressions in the different continuous assignment statements, then the synthesizer flashes an error “Multiple Driver Assignment”. Similarly if same ‘reg’ variable is driven by the different expressions in different always block, then it is multiple driver error. Exception for this is tri-state.

Consider an Example 4.17. In this example net y\_tmp is driven by two different expressions coded by using multiple ‘assign’.

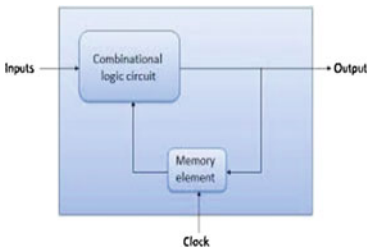
## 4.16 Summary

As discussed in this chapter following are important design guidelines

1. Use blocking assignments for design of combinational logic.
2. All the blocking assignments are evaluated and updated in the active event queue.
3. Use ‘case-endcase’ to infer parallel logic and use ‘if-else’ to infer priority logic
4. Use all the case conditions or ‘default’ in the ‘case-endcase’ to avoid unintentional of latches.
5. Use all the required inputs or signals in the sensitivity list of ‘always’ block. This is recommended to avoid simulation and synthesis mismatch.
6. Avoid use of multiple assignments to same net while using ‘assign’. This is recommended to avoid the multiple driver assignment error.
7. Avoid use of combinational looping as it exhibits the oscillatory behavior.
8. Cover all the ‘case’ conditions and ‘else’ conditions as missing ‘case’ conditions or ‘else’ conditions infers the unintentional latches in the design.

# Chapter 5

## Sequential Logic Design



An RTL design is incomplete without any register logic. RTL is Register Transfer Level or logic and used to describe the digital logic dependent on the present input and past output. This chapter discusses about the efficient RTL Verilog coding for latches, flip-flops, counters and shift-registers etc.

**Abstract** This chapter describes the detail practical understanding about the sequential logic designs. RTL coding using Verilog is described in detail with the practical scenarios and concepts. The Verilog RTL for the flip-flops, latches, various counters, shift registers, and memories is covered with the synthesized results and explanations. The practical do's and don'ts are explained with the meaningful diagrams and timing sequences. This chapter will be useful for the ASIC designers while coding for the sequential logic. This chapter also covers the necessity of registered input and register outputs.

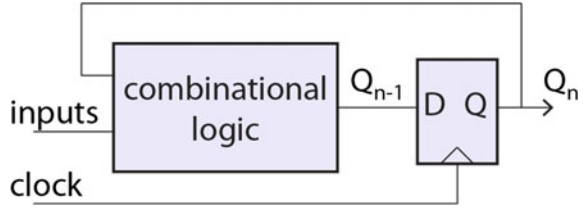
**Keywords** Latch · Flip-flop · Edge triggered · Level sensitive · Asynchronous · Synchronous · Toggle · Cumulative delay · Up-down · Shift register · Ripple · Johnson · Ring · Register input · Register output · Memory · Performance · Timing analysis · Glitch · Spike

### 5.1 Sequential Logic

Sequential logic is defined as the digital logic whose output is a function of present input and past output. So the sequential logic holds the binary data. Sequential logic elements are latches and flip-flops and are used to design the sequential circuits for



**Fig. 5.1** Basic sequential logic

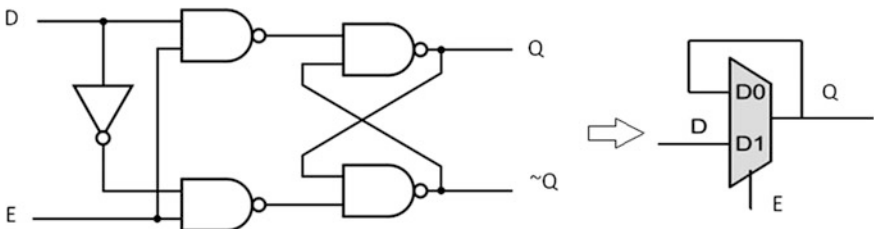


the given design functionality. For the RTL design engineer it is essential to understand the efficient RTL design for clocked-based logic circuits. The sequential logic elements are used to hold a larger amount of data in the complex designs. The logic is triggered on the clock. The subsequent chapter discusses on the efficient Verilog RTL to describe the required sequential logic. In the practical applications, it is always essential to describe the logic circuit which can be triggered on either positive edge of clock or on the negative edge of clock. It is always expected that the designed circuit should generate the finite output for finite duration of clock period. Figure 5.1 describes the basic sequential logic triggered on positive edge of clock. The output from the logic is a function of a present input and past output.

**5.1.1 Positive Level Sensitive D-Latch**

Latches are sensitive to the level. In the D-latch, D stands for the data input. The latches are either sensitive to positive or negative level of clock or enable. Positive level sensitive latch is shown in Fig. 5.2 and truth table is described in Table 5.1. As shown in Table 5.1 for latch enable 'E' is equal to positive level (logical '1') output, Q is equal to data input 'D' else output remains in the previous state (past output) and is shown by  $Q_{n-1}$ . The timing sequence is shown in Fig. 5.3.

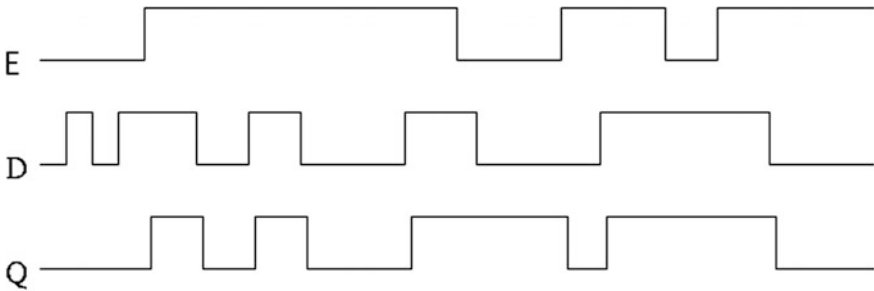
From the timing sequence it is clear that the output 'Q' is equal to data input 'D' during the time period for which enable input 'E' is equal to positive level. So D latch acts transparent during this period. During negative level (logical '0') of enable 'E', D latch holds the previous value.



**Fig. 5.2** Positive level sensitive D latch

**Table 5.1** Truth table for positive level sensitive D latch

E	D	Q	$\sim Q$
1	0	0	1
1	1	1	0
0	X	$Q_{n-1}$	$\sim Q_{n-1}$



**Fig. 5.3** Timing sequence for positive level sensitive D latch

Now the important point in the mind of the readers is how to describe positive level sensitive D latch using Verilog. It is very simple to visualize and to describe the design functionality. Example 5.1 describes the Verilog RTL for the positive level sensitive D latch and the synthesizable hardware is shown in Fig. 5.4.

```
// Verilog RTL code for positive level sensitive D Latch
module positive_level_d_latch( D, LE, Q);
input D;
input LE;
output Q;
reg Q;
//Functionality of D-Latch
always@(D or LE)
begin
    if(LE)
        Q <= D;
end
endmodule
```

Positive level sensitive latch is described using the 'always' procedural block. For changes in the 'D' or 'LE' an 'always' block invokes and assigns the 'D' input to 'Q' output. As described in the functionality 'else' clause is missing so it generates latch.



**Example 5.1** Synthesizable RTL for positive level sensitive D latch

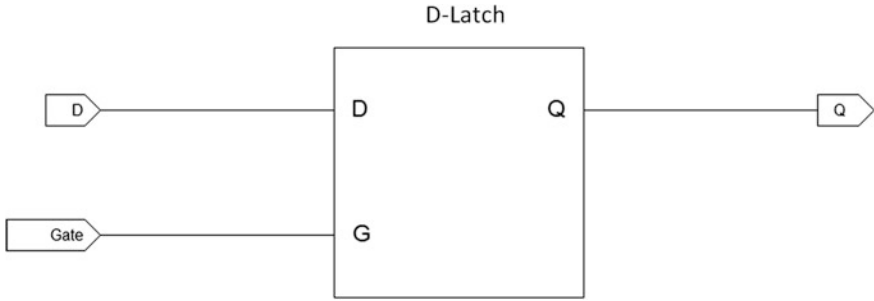


Fig. 5.4 Positive level sensitive D latch

### 5.1.2 Negative Level Sensitive D Latch

The truth table of the negative level sensitive D latch is described in Table 5.2 and it has active low or negative level sensitive latch enable ('LE\_n'), data input 'D', and output 'Q'.

The equivalent gate level representation is shown in Fig. 5.5. The latch acts as transparent on negative level of 'LE\_n' and holds the data during the positive level of 'LE\_n'. The timing sequence is shown in Fig. 5.6.

The Verilog RTL description is shown in Example 5.2 and the synthesized hardware is shown in Fig. 5.7.

Table 5.2 Truth table for negative level sensitive D latch

LE_n	D	Q	~Q
0	0	0	1
0	1	1	0
1	X	Q <sub>n-1</sub>	~Q <sub>n-1</sub>

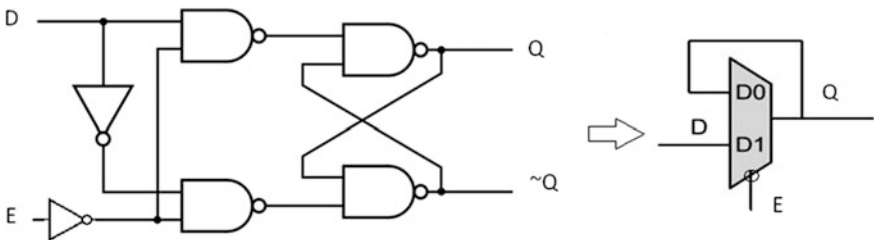


Fig. 5.5 Negative level sensitive D latch

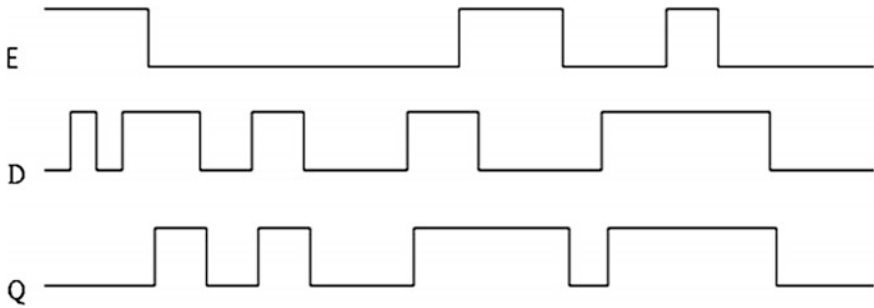


Fig. 5.6 Timing sequence for negative level sensitive latch

```
// Verilog RTL code for negative level sensitive D Latch
module negative_level_d_latch( D, LE_n, Q);
input D;
input LE_n;
output Q;
reg Q;
//Functionality of D-Latch
always@(D or LE_n)
begin
    if(~LE_n)
        Q <= D;
end
endmodule
```

Negative level sensitive latch is described using the 'always' procedural block. For changes in the 'D' or 'LE\_n' an 'always' block invokes and assigns the 'D' input to 'Q' output. As described in the functionality 'else' clause is missing so it generates latch.



Example 5.2 Synthesizable verilog RTL for negative level sensitive D latch

## 5.2 Flip-Flop

Flip-flop is an edge triggered logic circuit. It can be triggered either on positive edge of clock or on negative edge of clock. Flip-flop can be realized by using positive and negative level sensitive latches in cascade. Flip-flop is used as a memory storage element. Flip-flops are set-reset (SR), JK, D, and toggle. In an ASIC design

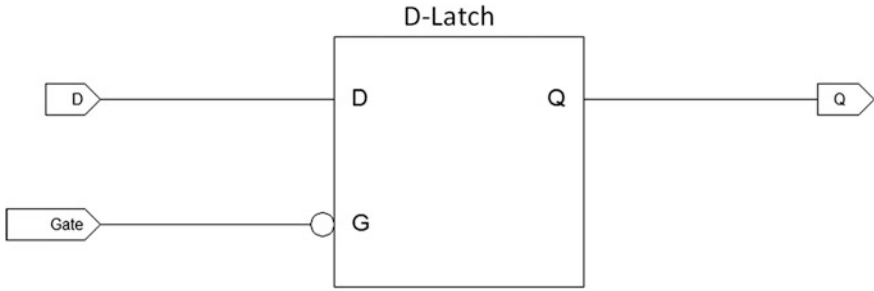


Fig. 5.7 Synthesized hardware for negative level sensitive latch

the D flip-flop is used as a sequential circuit element, where D stands for the data input. The subsequent section discusses on the positive and negative edge triggered flip-flop.

**5.2.1 Positive Edge Triggered D Flip-Flop**

Positive edge triggered D flip-flop is triggers on positive edge on clock. Practically there is no logic gate which can be triggered on edge! Positive edge triggered D flip-flop can be visualized as connection of negative level sensitive latch followed by positive level sensitive latch. The logic circuit for positive edge triggered D flip-flop is shown in Fig. 5.8.

**5.2.2 Negative Edge Triggered D Flip-Flop**

Negative edge triggered D flip-flop is triggers on negative edge of clock. Negative edge triggered D flip-flop can be visualized as connection of positive level sensitive latch followed by negative level sensitive latch. The logic circuit for positive edge triggered D flip-flop is shown in Fig. 5.9.

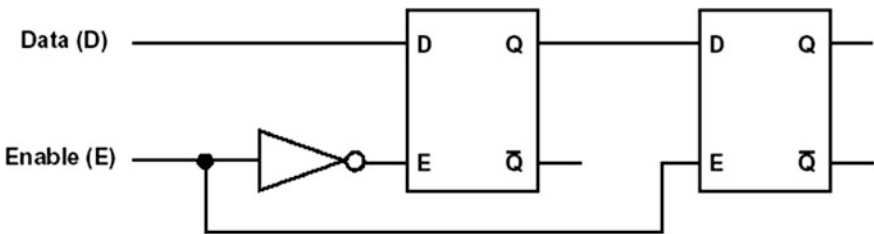
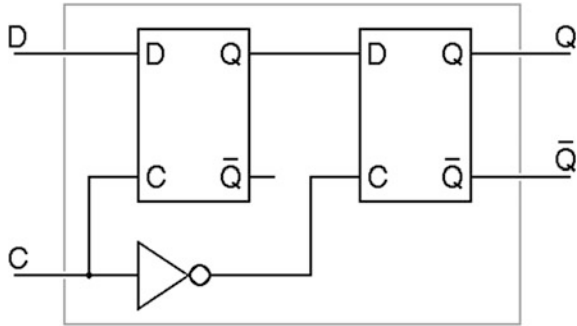


Fig. 5.8 Positive edge triggered D flip-flop

**Fig. 5.9** Negative edge triggered D flip-flop



### 5.3 Synchronous and Asynchronous Reset

In a ASIC design, when to use asynchronous reset or synchronous reset always leads to confusion in the mind of designer. Synchronous reset signal is sampled on clock edge and part of the data path, whereas, asynchronous reset signal is sampled irrespective of clock signal and not a part of data path or data input logic. This section describes about Verilog RTL for D flip-flop using asynchronous and synchronous reset.

#### 5.3.1 D Flip-Flop Asynchronous Reset

Asynchronous reset is not a part of data path and used to initialize flip-flop irrespective of active clock edge of clock and hence, named as asynchronous reset. This technique to initialize flip-flop is not recommended for internal reset signal generation as it is prone to glitches. Care needs to be taken by designer to synchronize this reset signal internally to avoid the glitches. The internally synchronized reset signal is applied to the storage elements. The reset deassertion is the main problem in the asynchronous reset signals and this problem can be overcome by using two stage level synchronizer. Level synchronizer avoids the race around conditions during reset deassertion.

Verilog RTL is shown in the figure and uses active low asynchronous reset signal 'reset\_n' (Example 5.3).

The synthesized logic for D flip-flop with asynchronous reset 'reset\_n' is shown in Fig. 5.10.

```

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

input d_in;

input clk;

input reset_n;

output reg q_out;

always@(posedge clk or negedge reset_n)

begin

    if(~reset_n)

        q_out <= 1'b0;

    else

        q_out <= d_in;

end

endmodule
    
```



An 'always' procedural block is sensitive to positive edge of clock 'clk' or to negative edge of reset 'reset\_n'.

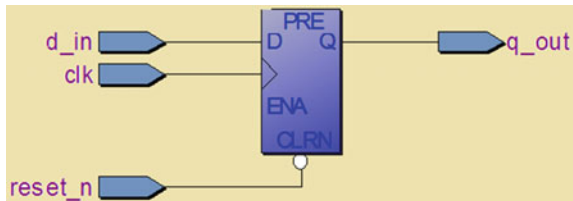
As negative edge of 'reset\_n' is included in the sensitivity list; it generates D flip-flop with asynchronous reset.

For reset 'reset\_n=0', output q\_out is assigned to logic '0'

For 'reset\_n=1' clock. output 'q\_out' is assigned to value d\_in on positive edge of clock 'clk'

**Example 5.3** D flip-flop with asynchronous active low reset input

**Fig. 5.10** Synthesized D flip-flop with asynchronous active low reset input



### 5.3.2 D Flip-Flop Synchronous Reset

In synchronous reset, the reset signal is part of data input that is data path and depends upon the active clock edge. The synchronous reset does not have issues of glitches or hazards so this approach is best suited for the design. This mechanism does not require the additional synchronization circuit.

Verilog RTL is described in Example 5.4 and uses active low synchronous reset signal 'reset\_n'.

```

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

input d_in;

input clk;

input reset_n;

output reg q_out;

always@(posedge clk)

begin

    if(~reset_n)

        q_out <= 1'b0;

    else

        q_out <= d_in;

end

endmodule

```



An 'always' procedural block is sensitive to positive edge of clock 'clk'.

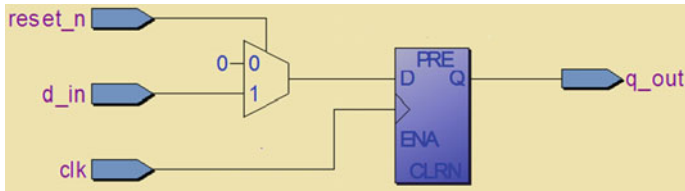
As 'reset\_n' is not included in the sensitivity list; it generates D flip-flop with synchronous reset. Reset 'reset\_n' is sampled on every positive edge of clock 'clk'.

For reset 'reset\_n=0', output q\_out is assigned to logic '0'

For 'reset\_n=1' clock, output 'q\_out' is assigned to value 'd\_in' on positive edge of clock

**Example 5.4** D flip-flop with active low synchronous reset input





**Fig. 5.11** Synthesized logic for D flip-flop with synchronous reset

The synthesized logic for positive edge triggered D flip-flop with synchronous reset input is shown in Fig. 5.11.

### 5.3.3 Flip-Flop with Load Enable Asynchronous Reset

In most of the practical applications multiple asynchronous inputs are required. Consider an application where it is required to load the input data when enable input is active. Even it is essential to initialize register when reset signal is active and valid. If both asynchronous inputs arrive at a time then the output should be dependent on the priority assignment for these signals.

As shown in Example 5.5, two asynchronous inputs are named as ‘reset\_n’ and ‘load\_en’. The ‘reset\_n’ has highest priority and ‘load\_en’ has the lowest priority. The priority is scheduled using ‘if-else’ construct.

The synthesized logic is shown in Fig. 5.12.

### 5.3.4 Flip-Flop with Synchronous Load and Synchronous Reset

If multiple signals or inputs are part of the data path and sampled on the active edge of clock then output of sequential cell is assigned on the active edge of clock. Consider the Verilog RTL shown in Example 5.6, inputs ‘reset\_n’, and ‘load\_en’ are synchronous inputs and sampled on the positive edge of the clock. Synchronous input ‘reset\_n’ has highest priority and ‘load\_en’ has the lowest priority.

The synthesized logic is shown in Fig. 5.13 and ‘reset\_n’ and ‘load\_en’ are part of data paths.

```

module d_flip_flop( d_in, load_en, clk, reset_n, q_out);

input d_in;

input load_en;

input clk;

input reset_n;

output reg q_out;

always@(posedge clk or negedge reset_n)

begin
    if(~reset_n)
        q_out <= 1'b0;

    else if (load_en)
        q_out <= 1'b1;

    else

        q_out <= d_in;

end

endmodule
    
```



An 'always' procedural block is sensitive to positive edge of clock 'clk' or to negative edge of reset 'reset\_n'.

As negative edge of 'reset\_n' is included in the sensitivity list; it generates D flip-flop with asynchronous reset.

For reset 'reset\_n=0', output q\_out is assigned to logic '0' and has high priority.

Another input 'load\_en' acts as an enable input and used to set the output 'q\_out' of flip-flop to logic '1' and has second priority after reset.

For 'reset\_n=1' clock, output 'q\_out' is assigned to value d\_in on positive edge of clock 'clk'.

**Example 5.5** Verilog RTL of D flip-flop with asynchronous load and reset

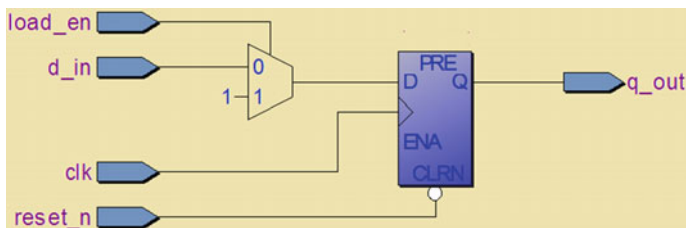


Fig. 5.12 Synthesized output for D flip-flop with asynchronous reset

## 5.4 Synchronous Counters

If all the storage elements are triggered by same source clock signal then the design is said to be synchronous. The advantage of synchronous design is the overall propagation delay for the design is equal to propagation delay of flip-flop or storage element. STA is very easy for the synchronous logic and even the performance improvement is possible by using the pipelining. Most of the ASIC implementation uses the synchronous logic. This section describes the synchronous counter design.

Four-bit binary counter is used to count from '0000' to '1111' and the four-bit BCD counter is used to count from the '0000' to '1001'. Figure 5.14 shows the four-bit binary counter where every stage is divided by two counters.

As shown in Fig. 5.14. The counter has four output lines 'QA, QB, QC, QD' where 'QA' is LSB and 'QD' is MSB. The output at 'QA' toggles on every clock pulse and hence divided by two. Output at 'QB' toggles for every two clock cycles and hence it is divisible by four, at 'QC' output toggles for ever four clock cycles and hence the output is divided by eight. Similarly the output at 'QD' toggles for every eight cycle and hence output at 'QD' is divided by sixteen of the input clock time period. In the practical applications counters are used as clock divider network. Even counters are used in the frequency synthesizers to generate variable frequency outputs.

### 5.4.1 Three Bit Up Counter

Counters are used to generate the predefined and required count sequence on the active edge of clock. In ASIC design it is essential to write an efficient RTL code for the counter by using the synthesizable constructs. Three-bit up counter is described by using Verilog to generate synthesizable design. Counter counts from '000' to '111' on the positive edge of the clock and wraps around to '000' on the next positive edge of the count. The counter described in Example 5.7 is presettable

```
module d_flip_flop( d_in, load_en, clk, reset_n, q_out);

input d_in;
input load_en;

input clk;

input reset_n;

output reg q_out;

always@(posedge clk)

begin

    if(~reset_n)

        q_out <= 1'b0;

    else if (load_en)

        q_out <= 1'b1;

    else

        q_out <= d_in;

end

endmodule
```

An 'always' procedural block is sensitive to positive edge of clock 'clk'.

As 'reset\_n' is not included in the sensitivity list; it generates D flip-flop with synchronous reset and synchronous load.

For reset 'reset\_n=0', output q\_out is assigned to logic '0' and has high priority.

Another input 'load\_en' acts as an enable input and used to set the output 'q\_out' of flip-flop to logic '1' and has second priority after reset.

For 'reset\_n=1' counter output 'q\_out' is assigned to value d\_in on positive edge of clock 'clk'

**Example 5.6** D flip-flop with synchronous load\_en and synchronous reset\_n

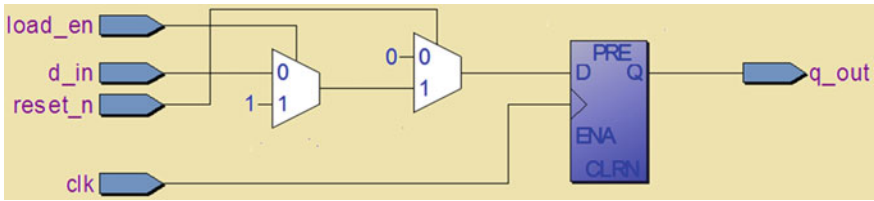


Fig. 5.13 Synthesized logic with synchronous reset\_n and synchronous load

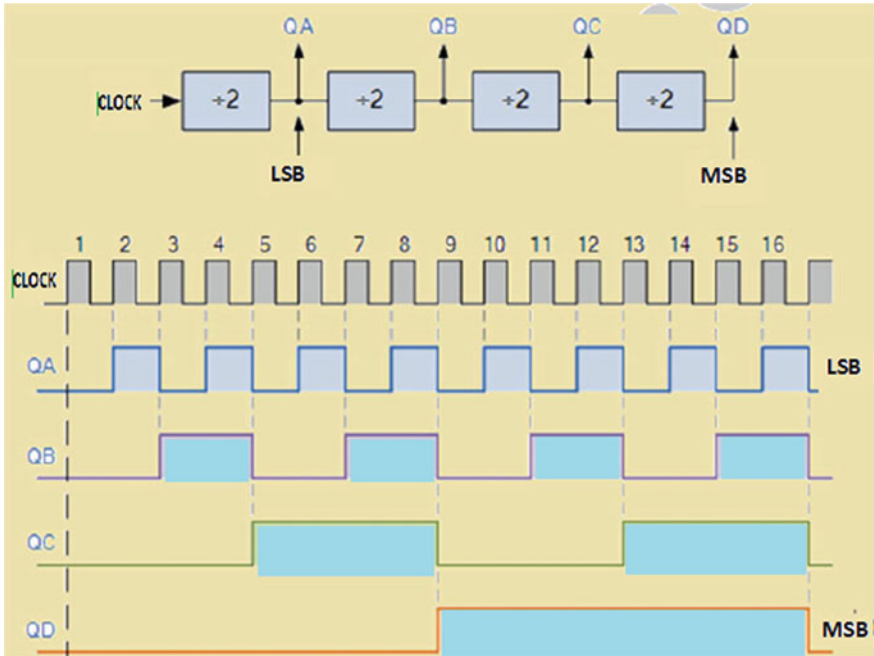


Fig. 5.14 Four-bit binary counter

counter and it has the synchronous active high 'load\_en' input to sample the three-bit required presettable value. The data input is three bit and indicated as 'data\_in'.

Counter has active low asynchronous 'reset\_n' input and when it is active low the status on output line 'q\_out' is '000'. During normal operation 'reset\_n' is active high.

The synthesizable output is shown in Fig. 5.15 and has three bit data input lines 'data\_in', active high 'load\_en', and active low reset input 'reset\_n'. Output is indicated by the 'q\_out' lines and positive edge triggered clock by 'clk'.

```

module up_counter_3bit (data_in, load_en, clk, reset_n, q_out);

input [2:0] data_in;

input load_en;

input clk;

input reset_n;

output reg [2:0] q_out;

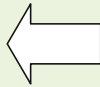
always@(posedge clk or negedge reset_n)
begin
    if(~reset_n)
        q_out <= 3'b000;

    else if (load_en)
        q_out <= data_in;
    else
        q_out <= q_out +1'b1;

end

endmodule

```



Procedural 'always' block is sensitive to clock 'clk', reset 'reset\_n'.

For reset 'reset\_n=0' it assigns 3-bit output 'q\_out=000'. Here reset is asynchronous input and has highest priority over any other input.

For load signal 'load\_en=1' the input 'data\_in' is assigned to output 'q\_out'

For 'reset\_n=1' and 'load\_en=0' the else clause is executed and increments the counter output 'q\_out' on every positive edge of clock.

**Example 5.7** Verilog RTL for three-bit up counter

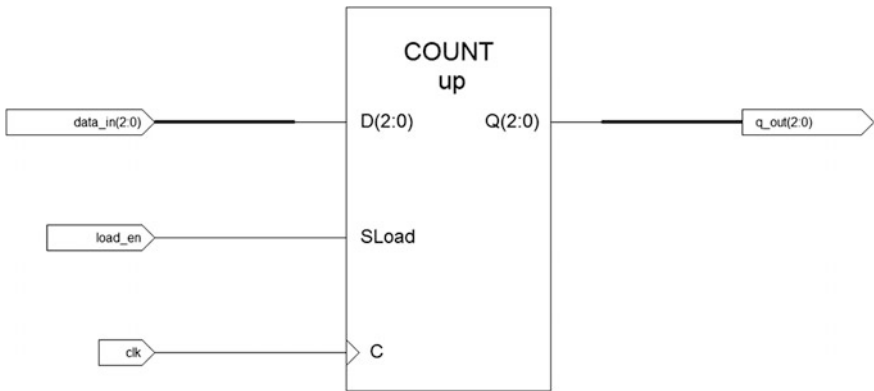


Fig. 5.15 Synthesized three-bit up counter top-level diagram

### 5.4.2 Three-Bit Down Counter

Three-bit down counter is described by using Verilog to generate synthesizable design. Counter counts from '111' to '000' and is triggered on the positive edge of the clock and wraps around to '111' on the next positive edge of the count after reaching to count value '000'. The timing sequence for the three-bit down counter is shown in Fig. 5.16.

The counter described in Example 5.9 is presettable counter and it has the synchronous active high 'load\_en' input to sample the three bit required presettable value. The data input is three bit and indicated as 'data\_in'.

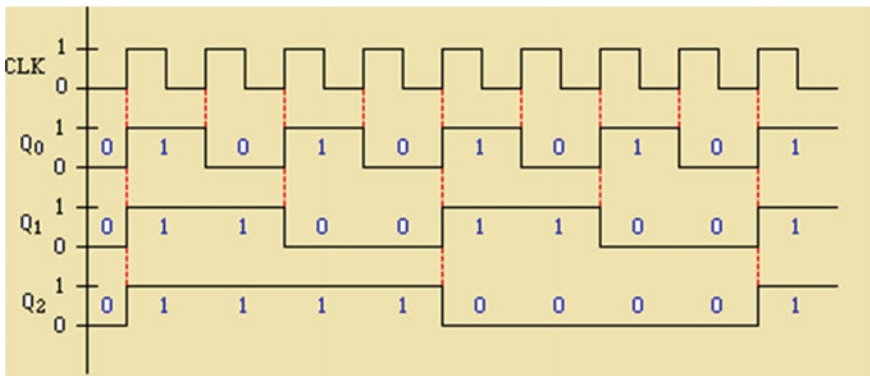


Fig. 5.16 Timing sequence for three-bit binary down counter

```

module down_counter_3bit (data_in, load_en, clk, reset_n, q_out);

input [2:0] data_in;

input load_en;

input clk;
input reset_n;

output reg [2:0] q_out;

always@(posedge clk or negedge reset_n)

begin

    if(~reset_n)

        q_out <= 3'b000;

    else if (load_en)

        q_out <= data_in;

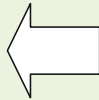
    else

        q_out <= q_out -1'b1;

end

endmodule

```



Procedural 'always' block is sensitive to clock 'clk', reset 'reset\_n'.

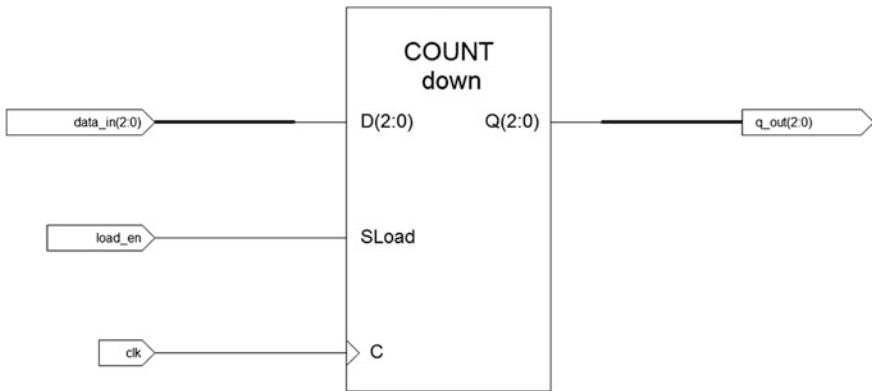
For reset 'reset\_n=0' it assigns 3-bit output 'q\_out=000'. Here reset is asynchronous input and has highest priority over any other input.

For load signal 'load\_en=1' the input 'data\_in' is assigned to output 'q\_out'

For 'reset\_n=1' and 'load\_en=0' the else clause is executed and decrements the counter output 'q\_out' on every positive edge of clock.

**Example 5.8** Verilog RTL for three-bit down counter





**Fig. 5.17** Synthesized three bit down counter top-level diagram

Counter has active low asynchronous 'reset\_n' input and when it is active low the status on output line 'q\_out' is '000'. During normal operation 'reset\_n' is active high.

The synthesizable output is shown in Fig. 5.17 and has three-bit data input lines 'data\_in', active high 'load\_en', and active low reset input 'reset\_n'. Output is indicated by the 'q\_out' lines and positive edge triggered clock by 'clk'.

### 5.4.3 Three-Bit Up-Down Counter

Three-bit up-down counter is described by using Verilog to generate synthesizable sequential design. Down counter counts from '111' to '000' and triggered on the positive edge of the clock and wraps around to '111' on the next positive edge of the count after reaching to count value '000'. Up counter counts from '000' to '111' and is triggered on the positive edge of the clock and wraps around to '000' on the next positive edge of the count after reaching to count value '000'.

Figure 5.18 gives the internal structure of three-bit binary up-down counter. For UP/DOWN is equal to logic '1' the counter acts as up counter and for UP/DOWN is equal to '0' counter acts as down counter.

The counter described in Example 5.9 is presettable counter and it has the synchronous active high 'load\_en' input to sample the three bit required presettable value. The data input is three bit and indicated as 'data\_in'. The up or down counting operation is selected by the input 'up\_down', for 'up\_down = 1' counter acts as up counter and for 'up\_down = 0' counter acts as down counter.

Counter has active low asynchronous 'reset\_n' input and when it is active low the status on output line 'q\_out' is '000'. During normal operation 'reset\_n' is active high (Example 5.9).

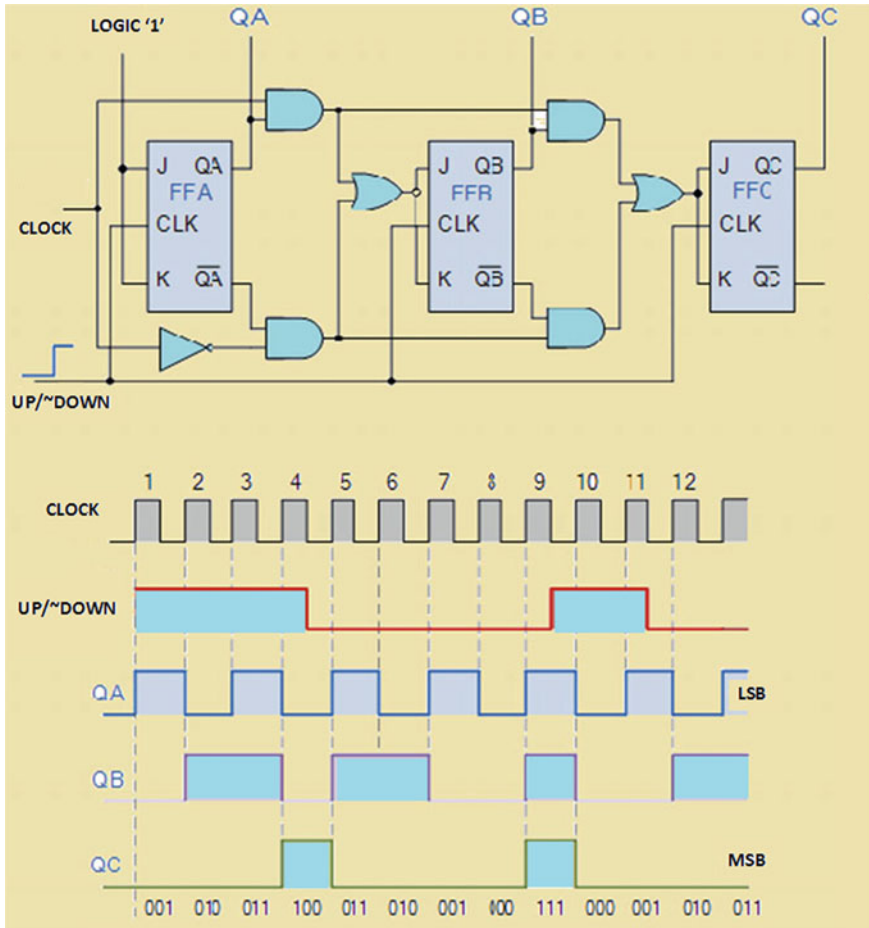


Fig. 5.18 Three-bit up-down counter

The synthesizable output is shown in Fig. 5.19 and has three-bit data input lines ‘data\_in’, active high ‘load\_en’, and active low reset input ‘reset\_n’. Output is indicated by the ‘q\_out’ lines and positive edge triggered clock by ‘clk’ and select line is ‘up\_down’.

### 5.4.4 Gray Counters

Gray counters are used in the multiple clock domain designs as only one bit changes on the active clock edge. Gray codes are used in the synchronizers. Gray

```

module up_down_counter_3bit (data_in, load_en, up_down, clk, reset_n,
    q_out);

input [2:0] data_in;

input load_en;

input up_down;

input clk;

input reset_n;

output reg [2:0] q_out;

always@(posedge clk or negedge reset_n)

begin

    if(~reset_n)
        q_out <= 3'b000;

    else if (load_en)

        q_out <= data_in;

    else

        begin

            if (up_down)

                q_out <= q_out +1'b1;

            else

                q_out <= q_out - 1'b1;

        end

end

endmodule

```

Procedural 'always' block is sensitive to clock 'clk', reset 'reset\_n'.

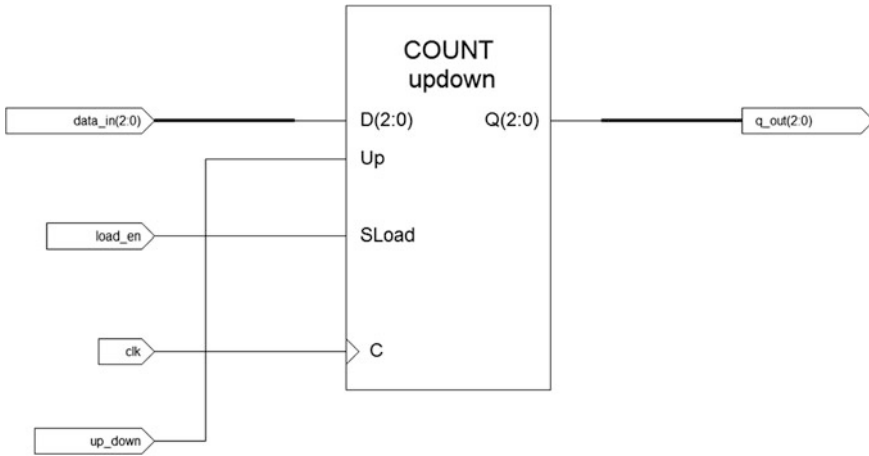
For reset 'reset\_n=0' it assigns 3-bit output 'q\_out=000'. Here reset is asynchronous input and has highest priority over any other input.

For load signal 'load\_en=1' the input 'data\_in' is assigned to output 'q\_out'

For 'reset\_n=1' and 'load\_en=0' the else clause is executed and increments or decrements the counter output 'q\_out' on every positive edge of clock.

For 'up\_down=1' counter increments by one and for 'up\_down=0' counter decrements by one.

**Example 5.9** Verilog RTL for three-bit up-down counter



**Fig. 5.19** Synthesized three-bit up-down counter top-level module

counter is described in the Example and in this only one bit is changing on the active clock edge with reference to the previous output of the counter. In this active high reset input is 'rst'. When 'rst = 1' then the output of counter 'out' is assigned to '0000'.

The counter described in Example 5.10 is presetable counter and it has the synchronous active high 'load\_en' input to sample the four-bit required presetable value. The data input is four bit and indicated as 'data\_in'.

Counter has active high asynchronous reset 'rst' input and when it is active high the status on output line 'out' is '0000'. During normal operation 'rst' is active low.

### 5.4.5 Gray and Binary Counter

In most of the practical applications binary and Gray counters need to be used. Gray counter output can be generated from the binary counter output by using the combinational logic. Refer Chap. 2 for the binary to Gray and Gray to binary code converters.

Parameterized binary and Gray counter is described in Example and the Verilog RTL is described to generate four-bit binary and Gray output. For 'rst\_n = 0' binary and Gray counter output is assigned to '0000'. Four-bit Gray code output is denoted as 'gray' (Example 5.11).

Simulation result for the four-bit binary counter is shown in the following timing sequence Fig. 5.20 and for every positive edge of clock counter output increments by one.

```
module gray_counter_design (clk, rst, data_in, load_en, out);
```

```
    input clk, rst;
```

```
    input load_en;
```

```
    input [3:0] data_in;
```

```
    output reg [3:0] out;
    reg q0, q1, q2;
```

```
    reg [3:0] count;
```

```
    always @ (posedge clk)
```

```
    begin
```

```
        if (rst)
```

```
            begin
```

```
                count <= 4'b0;
```

```
                {q2,q1,q0} <= 3'b000;
```

```
                out <= 4'b0;
```

```
            end
```

```
        else if (load_en)
```

```
            begin
```

```
                count <= data_in;
```

```
                {q2,q1,q0} <= { data_in [2], data_in[1], data_in [0]};
```

Four bit gray counter functionality is described by using procedural 'always' block and it is triggered on positive edge of clock 'clk'.

The counter has synchronous reset 'rst' and when it is active high the counter output 'out' is assigned to 'logic 0'.

For load input 'load\_en' four bit data input is loaded inside the counter.

For reset 'rst=0' and 'load\_en=0' counter is incremented by one and described by temporary register 'count'

Gray counter output is derived from the binary count 'count' value to get only one bit change in the successive count values.

Output of Gray counter is defined as 'out'.

### Example 5.10 Four-bit Gray counter

```
        out <= { data_in [3], data_in [2], data_in[1], data_in [0]};

    end

    else

begin

    count <= count + 1'b1;

    q2 <= count[3] ^ count[2];

    q1 <= count[2] ^ count[1];

    q0 <= count[1] ^ count[0];

    out <= {count[3], q2, q1, q0};

    end

end

endmodule
```

**Example 5.10** (continued)

### 5.4.6 Ring Counters

Ring counters are used in the practical applications to provide the predefined delay. These counters are synchronous in nature and used in the practical applications like traffic light controller, timers to introduce the certain amount of predefined delay. The internal logic structure using the D flip-flops for four-bit ring counter is shown in the Fig. 5.21, as shown the output of MSB flip-flop is fed back to the LSB flip-flop input and the counter shifts the data on every active edge of clock signal.

The Verilog RTL for the four-bit ring counter is described in Example 5.12, the counter has 'set\_in' input to set the input initialization value of '1000' and works on the positive edge of clock signal.

The synthesized logic is shown in Fig. 5.22.

```

module gray_counter #(parameter SIZE = 4)

(output reg [SIZE-1:0] gray,

input wire clk, full, inc, rst_n);

reg [SIZE-1:0] bin;

wire [SIZE-1:0] gnext, bnext;

always@(posedge clk or negedge rst_n)
if (!rst_n)

{bin, gray} <= 2'b00;

else

{bin, gray} <= {bnext, gnext};

assign bnext = !full ? bin + inc : bin;

assign gnext = (bnext>>1) ^ bnext;

endmodule

```



Three bit parameterized gray and binary counter is described with asynchronous reset 'rst\_n' and triggered on positive edge of clock 'clk'.

For reset 'rst\_n=0' the counter output is assigned as zero. Binary counter output is 'bin' and gray counter output is 'gray'.

The next value of binary counter and gray counter is evaluated by using two different continuous assignment statements.

**Example 5.11** Verilog RTL for parameterized binary and Gray counter

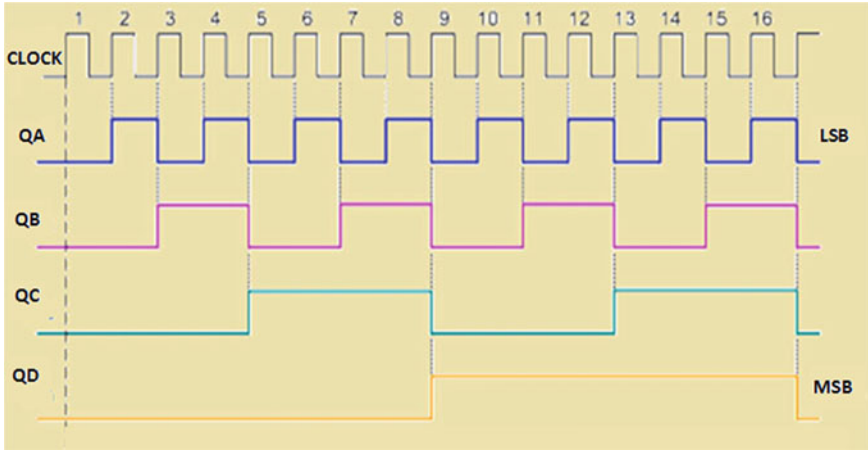


Fig. 5.20 Timing sequence for four-bit binary counter

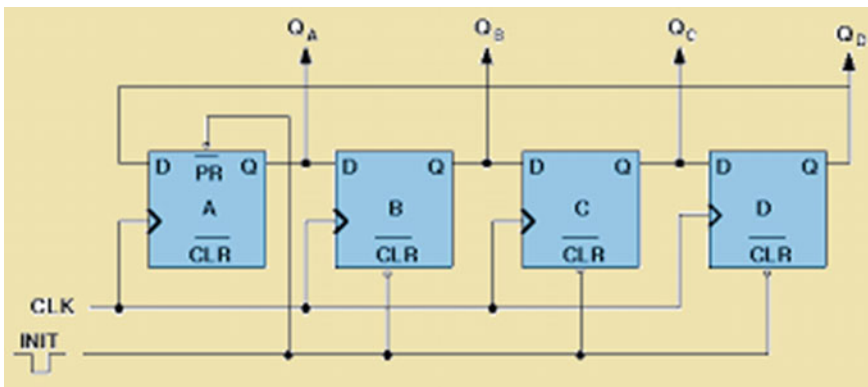


Fig. 5.21 Ring counter internal structure

### 5.4.7 Johnson Counters

The Johnson counter is the special type of synchronous counter and designed by using the shift register. The internal structure for three-bit Johnson counter is shown in Fig. 5.23.

The Verilog RTL for four-bit Johnson counter is shown in Example 5.13.

The synthesized logic is shown in Fig. 5.24.



```

module ring_counter (clk, set_in, count_out);
input clk;
input set_in;
output [3:0] count_out;
reg [3:0] count_out;
always @ (posedge clk)
begin
if (set_in)
begin
count_out <= 4'b1000;
end
else
begin
count_out <= (count_out << 1);
count_out[0] <= count_out[3];
end
end
endmodule
    
```



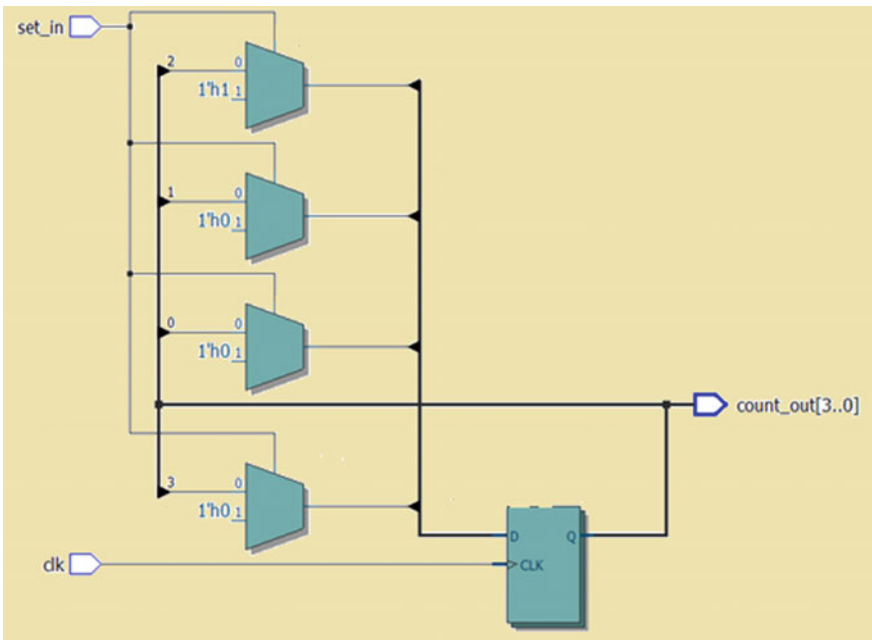
Ring counter is synchronous counter where the sequence is repeated after few clock cycles.

A synchronous input 'set\_in' is used to assign counter output 'count\_out' to four bit binary number "1000"

On positive edge of every clock signal counter output 'count\_out' is shifted by one bit and described by functionality  $count\_out <= (count\_out << 1)$ .

The MSB bit of counter is fed back to LSB flip-flop.

**Example 5.12** Verilog RTL for four-bit ring counter



**Fig. 5.22** Synthesized logic for four-bit ring counter

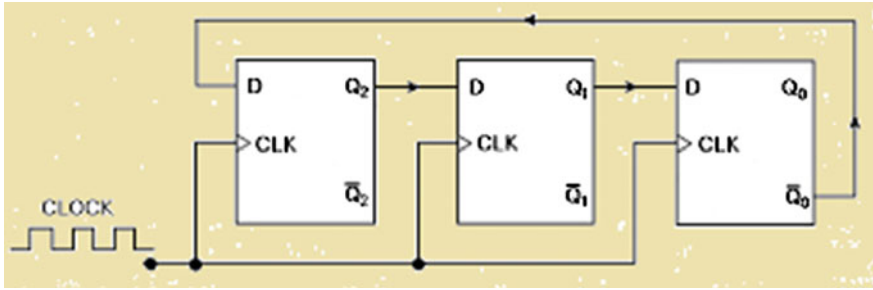


Fig. 5.23 Three-bit Johnson counter

```

module johnson_counter (reset_n, clock, q_out);
input reset_n;
input clock;
output reg [3:0] q_out;

always @( posedge clock , negedge reset_n )
if(~reset_n)
q_out<= 4'b0000;
else
q_out<= {{q_out[2:0]},~q_out[3]};
endmodule
    
```

Johnson counter is synchronous counter where the sequence is repeated after few clock cycles.

An asynchronous input 'reset\_n' is used to assign counter output 'q\_out' to four bit binary number "0000"

On positive edge of every clock signal counter output 'q\_out' is shifted by one bit. The complement of MSB bit of counter is fed back to LSB flip-flop.

Example 5.13 Verilog RTL for four-bit Johnson counter

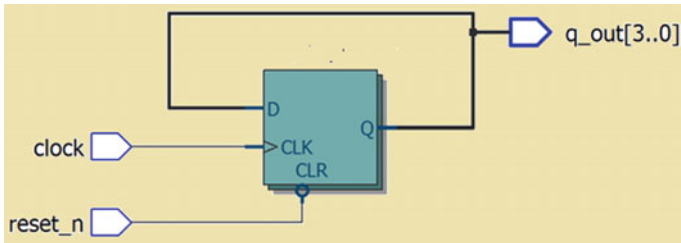


Fig. 5.24 Synthesized logic for four-bit Johnson counter

### 5.4.8 Parameterized Counter

In the practical applications to improve the readability and reusability the counters are designed by defining the parameter. The parameter integer value can be used to define the number of bits for the counter.

The Verilog RTL for the eight-bit parameterized counter is shown in Fig. 5.25.

The synthesizable top-level module for the parameterized counter is shown in Fig. 5.26.

## 5.5 Shift Register

Shift registers are used in most of the practical applications to perform the shifting or rotation operations on the active edge of clock. The shifter timing sequence with reference to the positive edge of clock signal is shown in Fig. 5.27. As shown in the timing sequence for every positive edge of the clock the data from LSB shifts by one bit to the next stage and hence, for the four-bit shift register it requires four clock latency to get the valid output data from MSB.

The Verilog RTL for the serial input serial output shift register is described in Example 5.14. As described in the example the data 'd\_in' is shifted on every clock edge to generate the serial output 'q\_out'. During normal operation reset input 'reset\_n' is set to logic '1'. To generate valid serial output for any change on the serial input the shift register needs four clock pulses.

The synthesized logic with four registers for the serial input serial output shift register is shown in Fig. 5.28.

### 5.5.1 Right and Left Shift

Most of the practical application involves the use of right or left shift of the data. Consider protocol which involves the processing of strings, where requirement is to shift the string on the right side or on the left side by one bit or by multiple bits. In such scenario the bidirectional (right/left) shift registers are used.

The Verilog RTL is described in Example 5.15 for bidirectional shift register and the direction of data is controlled by 'right\_left' input. For 'right\_left = 1' the data is shifted towards right side and for the 'right\_left = 0' the data is shifted towards left side.

The synthesized logic is shown in Fig. 5.29 and the direction of data transfer is controlled by 'right\_left' input. The synthesized logic consists of four registers with additional combinational logic to control, the data flow direction.

```

module parameterized_counter (data_in, load_en, reset_n, clock,
    count_out);

parameter N = 8;

input reset_n;

input clock;

input [N-1 : 0] data_in;

input load_en;

output [N-1:0] count_out;
reg [N-1:0] count_out;

always @ (posedge clock)

if (~reset_n)

count_out <= 0;

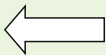
else if ( load_en)

count_out <= data_in;

else

count_out <= count_out + 1'b1;

endmodule
    
```



For parameterized counter the parameter value is configured to be 8 and hence act as an eight bit counter.

An 'always' procedural block is triggered on positive edge of clock and invokes the functionality for the counter.

Reset signal 'reset\_n' and load enable 'load\_en' are synchronous inputs.

Reset signal 'reset\_n' has highest priority and when 'reset\_n=0' an output 'count\_out' is assigned to zero.

For 'reset\_n=1' and 'load\_en=1' the counter output 'count\_out' is assigned to 'data\_in' input.

For 'reset\_n=1' and 'load\_en=0' the counter output 'count\_out' is incremented by one.

**Fig. 5.25** Verilog RTL for the eight-bit parameterized counter

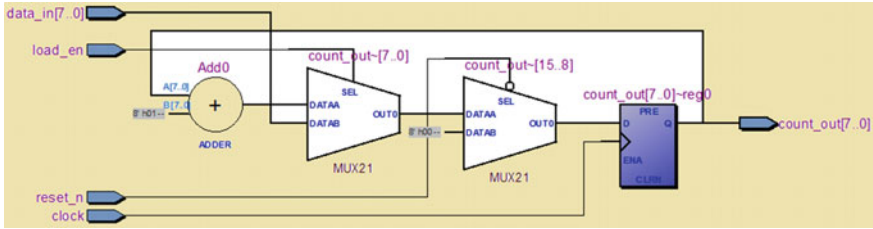


Fig. 5.26 Synthesized logic for parameterized counter

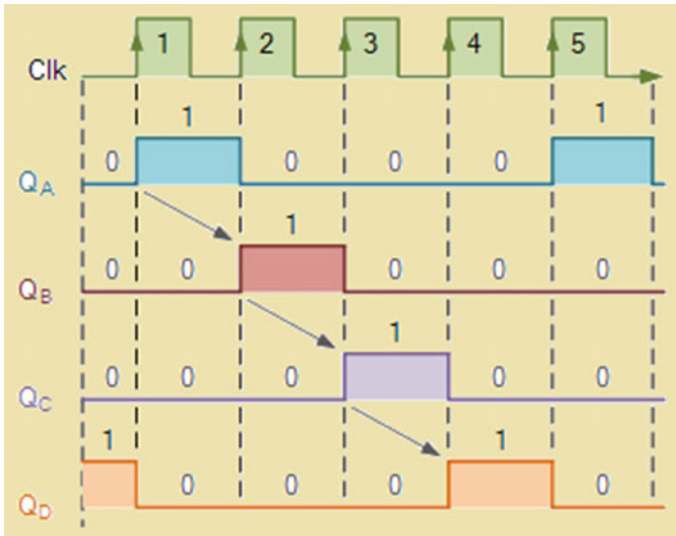


Fig. 5.27 Timing sequence of shift register

### 5.5.2 Parallel Input and Parallel Output (PIPO) Shift Register

In most of the processor design applications the data need to be transferred in parallel. Consider the four-bit data bus communicating with the external peripheral. If both processor and peripheral operates on the parallel data then it is essential to transfer the data using parallel input parallel output logic.

In such scenarios PIPO registers are used. The logic diagram of PIPO four-bit register is shown in Fig. 5.30. Four parallel input lines are named as  $P_A$ ,  $P_B$ ,  $P_C$ , and

```

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

input clk;

input reset_n;

input d_in;

output q_out;

reg q_out;

reg temp1_out, temp2_out, temp3_out;

always @ ( posedge clk or negedge reset_n)

begin

    if (~reset_n)

        begin

            temp1_out <= 1'b0;

            temp2_out <= 1'b0;

```



For asynchronous active low reset 'reset\_n=0' the output of all register is initialized to logic '0'.

Output from the MSB register 'q\_out' is initialized to zero.

During normal shift operations an input 'reset\_n' is set to be logic '1'

**Example 5.14** Verilog RTL for serial input serial output shift register

```

temp3_out <= 1'b0;

q_out <= 1'b0;

end

else

begin

temp1_out <= d_in;

temp2_out <= temp1_out;

temp3_out <= temp2_out;

q_out <= temp3_out;

end

end

endmodule

```

By using non-blocking assignment the functionality of serial input serial output shift register is described.

Four non-blocking assignments in the 'begin-end' block generate four bit shift register.

The shift register is triggered on positive edge of clock.

Example 5.14 (continued)

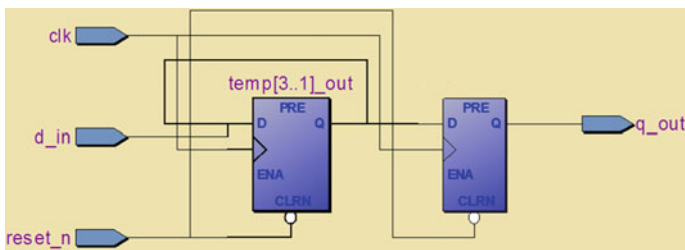


Fig. 5.28 Synthesized Logic for four-bit shift register

```
module right_left_shift_register( d_in, clk, reset_n, right_left,q_out);

input right_left;

input clk;

input reset_n;

input d_in;

output [3:0] q_out;

reg [3:0] q_out;

always @ ( posedge clk or negedge reset_n)

begin

if (~reset_n)

begin
q_out <= 4'b0000;

end

else

begin

if (right_left)

q_out <= { d_in, q_out[3:1] };

else

q_out <= { q_out[2:0], d_in };

end

end

endmodule
```



For 'right\_left=1' the data is shifted towards the right side by one-bit.

For the 'right\_left=0' the data is shifted towards the left side by one-bit.

**Example 5.15** Verilog RTL for the right/left shift register



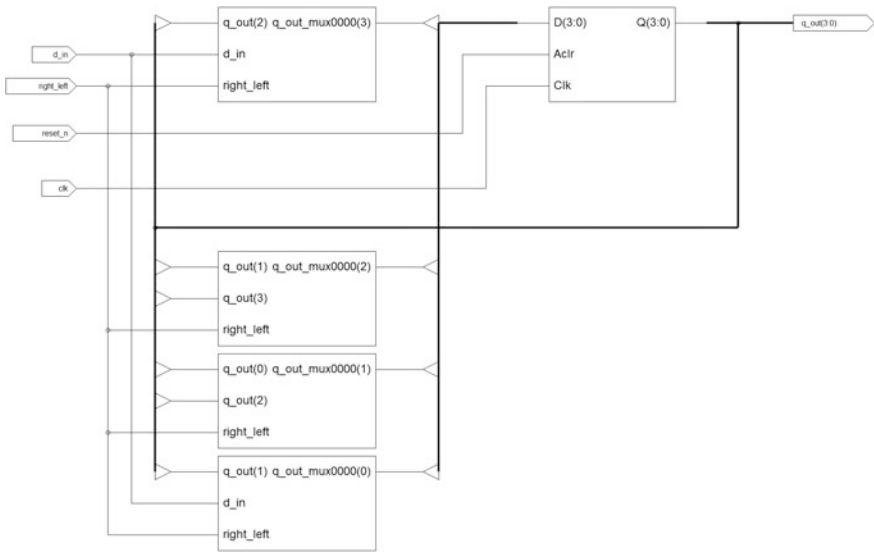


Fig. 5.29 Synthesized logic for bidirectional shift register

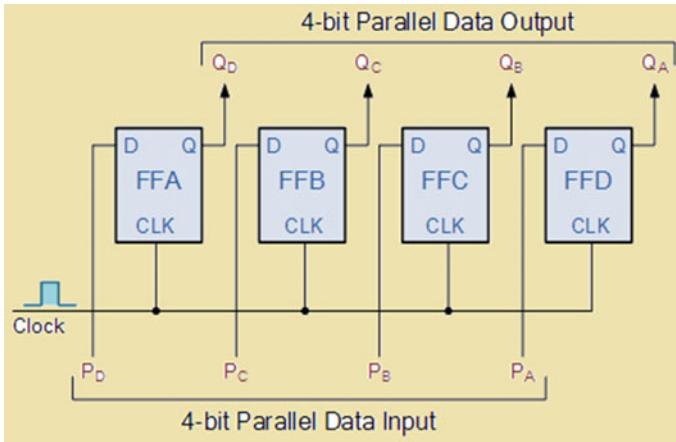


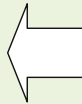
Fig. 5.30 Four-bit PIPO register

$P_D$  and four-bit parallel output lines are named as  $Q_A, Q_B, Q_C,$  and  $Q_D$ . The PIPO register is triggered on the positive edge of clock signal.

The Verilog RTL is described in Example 5.16.

The synthesized logic for the four-bit PIPO register is shown in Fig. 5.31.

```
module parallelin_parallelout( d_in, clk, reset_n, q_out);  
  
input clk;  
  
input reset_n;  
  
input [3:0] d_in;  
  
output [3:0] q_out;  
reg [3:0] q_out;  
  
always @ ( posedge clk or negedge reset_n)  
  
begin  
  
if (~reset_n)  
  
begin  
  
q_out <= 4'b0000;  
  
end  
  
else  
  
q_out <= d_in;  
  
end  
  
endmodule
```



For 'reset\_n=0' the PIPO register is initialized to '0000'

During normal operation PIPO register has 'reset\_n=1'.

On positive edge of the clock input the data input 'd\_in' is assigned to an output 'q\_out'.

As data input is four bit wide the logic inferred is four bit PIPO register.

**Example 5.16** Verilog RTL for 4-bit PIPO register

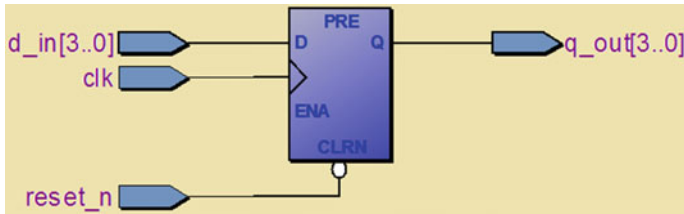


Fig. 5.31 Synthesized logic for four-bit PIPO register

## 5.6 Timing and Performance Evaluation

The timing is very important parameter for ASIC designs. Meeting timing for the sequential circuits is very crucial for the complex ASIC designs. The detail timing analysis and frequency calculations for the RTL designs will be discussed in the subsequent section.

For the better understanding of the same it is essential to have the oversight about the register inputs and register outputs. In the practical ASIC designs the Verilog code should be efficiently written and should have the register inputs and register outputs. The reason for the same is to have better timing analysis and to get the clean register to register paths.

The Verilog RTL with the register output is shown in Example 5.17. It is assumed that another module drives the input signals ‘a’, ‘b’, ‘c’, ‘d’, and ‘select’. All these inputs are registered inputs. This enables clean register path and easy timing analysis.

The synthesized logic is shown in Fig. 5.32 and generates the eight-bit parallel input parallel output register. The logic is triggered on the positive edge of clock.

## 5.7 Asynchronous Counter Design

In the asynchronous counters the clock signal is not driven by the common clock source. If the output of LSB flip-flop is given as an input to the subsequent flip-flop then the design is asynchronous. The issue with the asynchronous design is the cumulative clock to q delay of flip-flop due to the cascading of the stages. Asynchronous counters are not recommended in the ASIC design due to the issue of glitches or spikes and even the timing analysis for such kind of design is very complex.

```

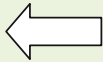
module mux_registered_output (a, b, c, d, y, select, clock);
  input [7:0] a, b, c, d;
  output [7:0] y;
  input [1:0] select;
  input clock;
  reg [7:0] y;

  always @ (posedge clock)

  case (select)
    0: y <= a; // non-blocking
    1: y <= b;
    2: y <= c;
    3: y <= d;
    default y <= 8'b0;
  endcase

endmodule

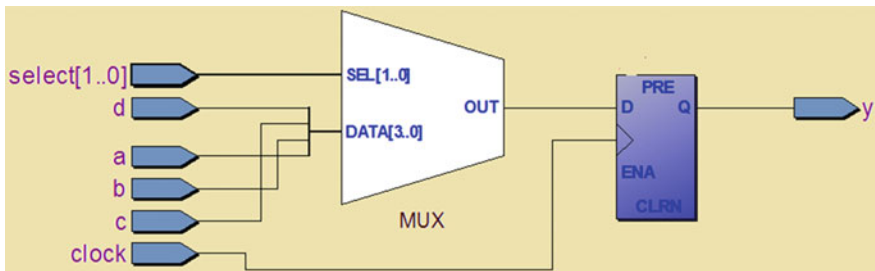
```



The 'case' construct is described inside the procedural 'always' block.

The procedural 'always' block is triggered on the positive edge of clock.

**Example 5.17** Verilog RTL for the register output



**Fig. 5.32** Synthesized logic for the register output logic

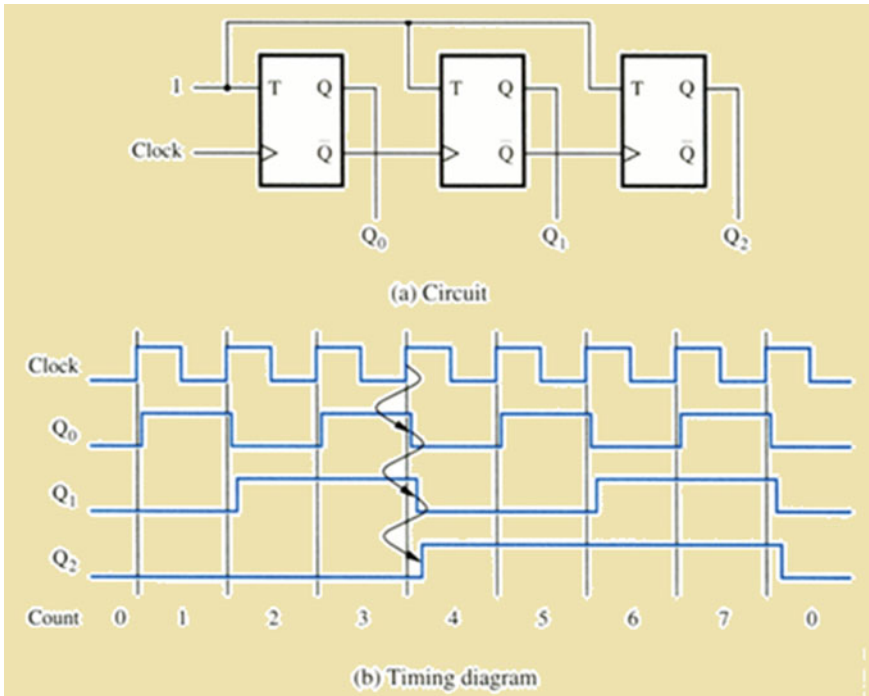


Fig. 5.33 Logic diagram of three bit ripple counter

### 5.7.1 Ripple Counters

The ripple counter is an asynchronous counter and shown in Fig. 5.33. As shown in the logic diagram all the flip-flops are positive edge triggered and the LSB register receives the clock from the master clock source. The output of LSB flip-flop is given as clock input to the next subsequent stage.

The Verilog RTL for the four-bit ripple up counter is shown in Example 5.18.

The synthesized logic is shown in Fig. 5.34.

## 5.8 Memory Modules and Design

In most of the ASIC designs and SOC-based designs memories are used to store the binary data. Memories can be of type ROM, RAM, single port, or dual port. The objective of this section is to describe basic single port read write memory. The timing sequence is shown in Fig. 5.35.

```

module ripple_counter (clock, toggle_in, reset_n, count_out);
  input clock, toggle_in, reset_n;
  output [3:0] count_out;
  reg [3:0] count_out;
  wire c0, c1, c2;
  assign c0 = count_out[0];
  assign    c1 = count_out[1];
  assign    c2 = count_out[2];

  always @ (negedge reset_n or posedge clock)
    if (reset_n == 1'b0)
      count_out[0] <= 1'b0;
    else if (toggle_in == 1'b1)
      count_out[0] <= ~count_out[0];

  always @ (negedge reset_n or negedge c0)
    if (reset_n == 1'b0)
      count_out[1] <= 1'b0;
    else if (toggle_in == 1'b1)
      count_out[1] <= ~count_out[1];

  always @ (negedge reset_n or negedge c1)
    if (reset_n == 1'b0)
      count_out[2] <= 1'b0;
    else if (toggle_in == 1'b1)
      count_out[2] <= ~count_out[2];

  always @ (negedge reset_n or negedge c2)
    if (reset_n == 1'b0)
      count_out[3] <= 1'b0;
    else if (toggle_in == 1'b1)
      count_out[3] <= ~count_out[3];
endmodule

```

Every 'always' block generates toggle flip-flop

Four 'always' blocks are used to describe the four single bit registers.



**Example 5.18** Verilog RTL for four-bit ripple up counter

As shown in the timing sequence the read write operation is controlled by 'rd\_wr' and data is sampled on the positive edge of the clock signal 'clk' if 'cs' is high. The address input is described by bus 'address'.

The Verilog RTL for the single port read write memory is shown in Example 5.19.

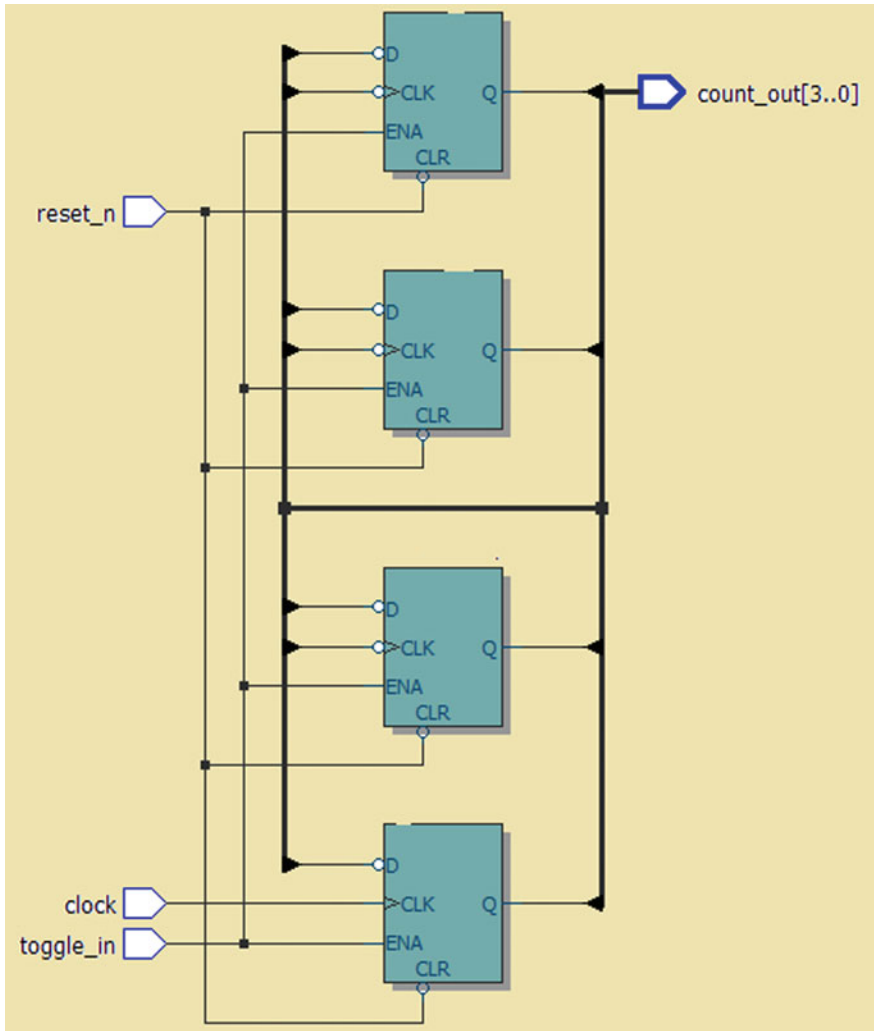
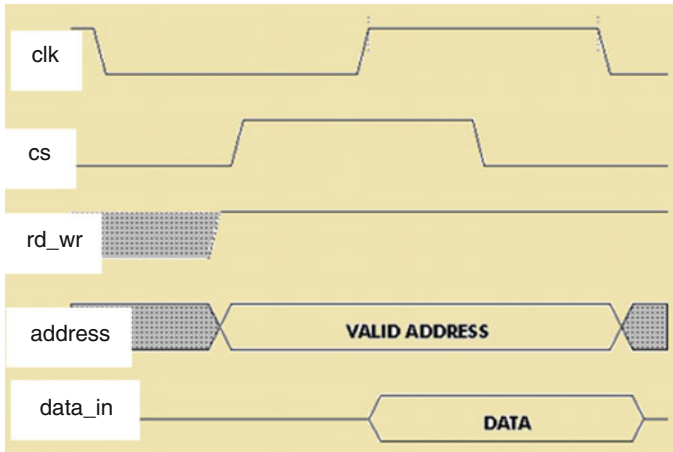


Fig. 5.34 Synthesized logic for four-bit ripple up counter



**Fig. 5.35** Timing sequence for the memory

```

module memory_read_write (clk, address, data_in, rd_wr, cs, da-
    ta_out);

input clk;
input [7:0] data_in;
input [9:0] address;
input rd_wr;
input cs;
output [7:0] reg data_out;
reg [7:0] memory [0:1023];

always@ (posedge clk)
if ( cs==1 && rd_wr==1)
memory [address] <= data_in;

always@ (posedge clk)
if ( cs==1 && rd_wr==0)
data_out <= memory [address];

endmodule
    
```

One 'always' block is used to perform the write operation when 'rd\_wr=1'.

Another 'always' block is used to perform the read operation when 'rd\_wr=0'



**Example 5.19** Verilog RTL for the read write memory



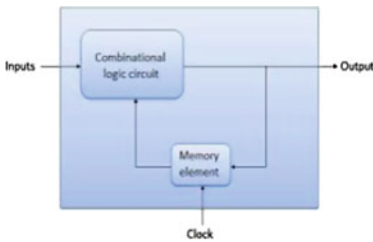
## 5.9 Summary

The following are the key points to summarize the sequential logic design.

1. Latches are level sensitive and not recommended in the ASIC designs.
2. Flip-flops are edge triggered and are recommended in the ASIC designs.
3. Flip-flops are described by using procedural block 'always' and triggered by either 'posedge clk' or 'negedge clk'.
4. Binary counters can be designed by using synchronous design concept or asynchronous design concept.
5. Gray counters can be designed by using the binary counters with the additional combinational logic.
6. Synchronous counters are recommended in the ASIC design as timing analysis will be easy and they are not prone to the glitches.
7. Asynchronous counters are prone to glitches or spikes and hence not recommended in the ASIC designs.
8. Special counters like ring and Johnson can be designed by using the shift registers.
9. The memories can be described by using the Verilog RTL to perform the read and write operation.

# Chapter 6

## Sequential Design Guidelines



An RTL design without sequential design guidelines can result into inefficient performance. This chapter discusses about the sequential design guidelines which need to be followed while writing an efficient Verilog RTL. Use of non-blocking assignments is described in detail while coding for the sequential logic designs.

**Abstract** This chapter describes about the key sequential design guidelines used in the ASIC design. These guidelines are essential for any ASIC design and used to improve the readability, performance, and need to be followed by an ASIC design engineer. The key guideline includes the use of nonblocking assignments in sequential designs, the use of synchronous resets and clock gating. The guidelines to use the pipelined stages in the design are described in detail and useful for improving the design performance. This chapter also covers the basic information about describing the Verilog RTL with multiple clocks, multiphase clocks and the issues with asynchronous resets.

**Keywords** Blocking · Nonblocking · Synchronous reset · Asynchronous reset · Cycle stealing · Time borrowing · Clock gating · Pipelining · Multiphase · Multiple clock domains · Data path · Control path

## 6.1 Use of Blocking Assignments

As discussed in Chap. 4, blocking assignments are recommended for describing the combinational logic designs. But what happens if blocking assignments are used while coding the sequential logic behavior? This is one of the most important questions need to be addressed and it is important for the subsequent discussion.

If blocking assignments are used for coding the behavior of sequential logic, then it is observed that the synthesized outcome is not the correct functional design intent.

This section describes the few design scenarios for the use of blocking assignments to code the sequential designs.

### 6.1.1 Blocking Assignments and Multiple “Always” Blocks

As described in Example 6.1, blocking assignments are used in the multiple “always” block. Procedural Block “always” is triggered on the positive edge of clock and synthesizer infers the sequential logic. As discussed already, all the blocking assignments are evaluated and updated in active queue. Readers are requested to refer Chap. 4 section stratified event queuing.

As described in Example 6.1, both “always” block executes in parallel and generates the output as two-bit serial in serial out shift register. First, always block generates an output “b\_in.” The output generated from the first “always” block is used as an input by another “always” block. Hence, synthesizer infers this as two-bit serial-input serial-output shift register.

Synthesized logic for Example 6.1 is shown in Fig. 6.1, and has input “a\_in,” “clk” and an output “y\_out.”

### 6.1.2 Blocking Assignments in the Same “Always” Block

If blocking assignments are used to describe the sequential logic and multiple assignments are used in the same “always” procedural block, then the desired intended requirement may or may not match with the synthesized logic. The reason being, in the blocking assignment is that all the trailing statements (next immediate) are blocked unless and until the present statement is executed. This results in truncation of the hardware and may infer the undesired synthesis output.

Consider the design scenario described in Example 6.2 and its intention is to create the three-bit serial-input and serial-output shift register but after synthesizing Example 6.2 it infers into the single flip-flop.

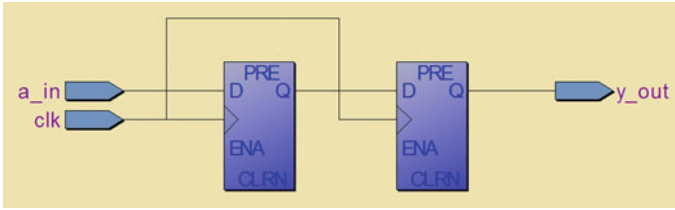
Synthesizable logic is shown in Fig. 6.2 which has inputs as ‘a’, ‘clk’, and an output ‘y’. The required functionality is serial-input serial-output shift register but

```
module blocking_assignments( a_in, clk, y_out );  
  
input wire a_in;  
  
input clk;  
  
output reg y_out; ←  
  
reg b_in;  
  
always @ ( posedge clk)  
  
begin  
  
    b_in = a_in;  
  
end  
  
always @ ( posedge clk) ←  
  
begin  
  
    y_out = b_in;  
  
end  
  
endmodule
```

Procedural 'always' block is triggered on positive edge of 'clk'  
On every positive edge of 'clk' 'a\_in' input value is assigned to intermediate variable 'b\_in'. This block infers positive edge triggered flip-flop with input 'a\_in' and output 'b\_in'

Procedural 'always' block is triggered on positive edge of 'clk'  
On every positive edge of 'clk' 'b\_in' input value is assigned to output 'y\_out'. This block infers positive edge triggered flip-flop with input 'b\_in' and output 'y\_out'

**Example 6.1** Blocking assignments in multiple always blocks



**Fig. 6.1** Synthesized logic for blocking assignments in multiple always block

```

module blocking_assignment(a,clk,y);
input a;
input clk;
output y;
reg y;
reg b,c;

always@(posedge clk)
begin
    b=a;
    c=b;
    y=c;
end
endmodule
    
```



Procedural 'always' block is triggered on positive edge of clock 'clk'.

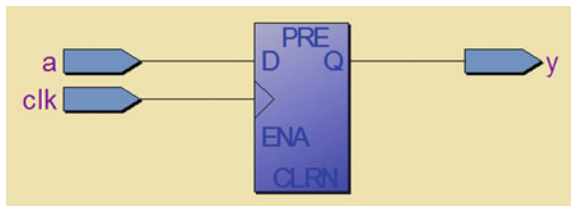
In between 'begin' and 'end' block, multiple blocking assignment statements are written.

The value of 'a' input is assigned to intermediate variable 'b' and so on.

Finally intermediate value 'c' is assigned to output 'y' and infers single register.

**Example 6.2** Blocking assignments in same always block

**Fig. 6.2** Synthesized logic for the blocking assignments in same always block



the above example infers the single flip-flop due to the use of blocking assignments. So, it is recommended to use the nonblocking assignments while coding or describing the RTL for the sequential functionality.

### 6.1.3 Example Blocking Assignment

Consider the design scenario described in Example 6.3 and its intention is to create the three-bit serial-input and serial-output shift register and due to the order of the blocking assignment statements used in the block “begin” and “end” it generates the three-bit serial-input serial-output shift register.

Synthesized logic is shown in Fig. 6.3 and has inputs as ‘a’, ‘clk’, and an output ‘y’. The required functionality is serial-input serial-output shift register and it infers the serial-input serial-output shift register. So, the important point to remember is that the order of the blocking assignment statement inside the procedural “always” block is decisive factor in the synthesis.

```

module blocking_assignment(a,clk,y);

input a;

input clk;

output y;
reg y;

reg b,c;

always@(posedge clk)

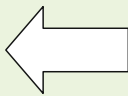
begin

    y=c;
    c=b;
    b=a;

end

endmodule

```



Procedural ‘always’ block is triggered on positive edge of clock ‘clk’.

In between ‘begin’ and ‘end’ block, multiple blocking assignment statements are written.

Finally intermediate variable value ‘c’ is assigned to output ‘y’ and infers single register.

The value of ‘b’ input is assigned to intermediate variable ‘c’ and so on and generates one bit register.

The value of an input ‘a’ is assigned to intermediate variable ‘b’ and generates one bit register.

**Example 6.3** Blocking assignments in the same always block (ordering)

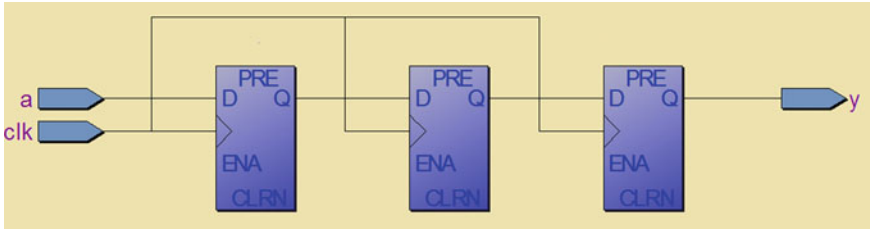


Fig. 6.3 Synthesizable logic after reordering of blocking assignments

## 6.2 Nonblocking Assignments

As discussed in Chap. 4, section “Stratified Event Queue”, nonblocking assignments are evaluated in the active event queue and updated in the NBA queue. Nonblocking assignments are used to describe the sequential logic. These assignments are used in the procedural block “always” to get the desired synthesis results. All the nonblocking assignments executes in parallel inside the “always” block.

As described in Example 6.4. Nonblocking assignments are used in the multiple “always” block. Procedural Block “always” is triggered on the positive edge of clock and synthesizer infers the sequential logic. The synthesized logic is shown in Fig. 6.4.

### 6.2.1 Example Nonblocking Assignment

If nonblocking assignments are used to describe the sequential logic and multiple assignments are used in the same “always” procedural block, then the desired intended logic is always inferred by synthesizer. The reason being, in the non-blocking assignment all the statements written in “begin-end” block are executed in parallel. This results in sequential logic.

Consider the design scenario described in Example 6.5, and intention is to create the three bit serial-input and serial-output shift register and nonblocking assignments are used.

Synthesized logic is shown in Fig. 6.5 and has inputs as ‘a’, ‘clk’, and an output ‘y’. The required functionality is serial-input serial-output shift register and it infers the serial-input serial-output shift register.

```
module non_blocking_assignments ( a_in, clk, y_out );  
  
input wire a_in;  
  
input clk;  
  
output reg y_out;  
  
reg b_in;  
  
always @ ( posedge clk)  
  
begin  
  
    b_in <= a_in;  
  
end  
  
always @ ( posedge clk)  
  
begin  
  
    y_out <= b_in;  
end  
  
endmodule
```

Procedural 'always' block is triggered on positive edge of 'clk'

On every positive edge of 'clk' 'a\_in' input value is assigned to intermediate variable 'b\_in'. This block infers positive edge triggered flip-flop with input 'a\_in' and output 'b\_in'

Another procedural 'always' block is triggered on positive edge of 'clk'

On every positive edge of 'clk' 'b\_in' input value is assigned to output 'y\_out'. This block infers positive edge triggered flip-flop with input 'b\_in' and output 'y\_out'

Example 6.4 Nonblocking assignments in the different always blocks

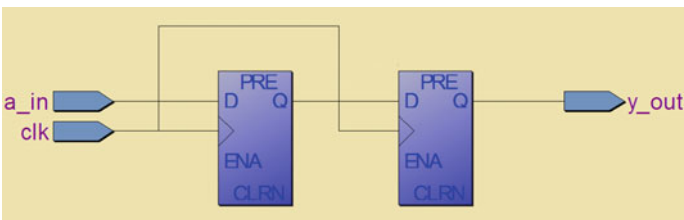


Fig. 6.4 Synthesized logic for nonblocking assignments in the different always blocks



```

module non_blocking_assignment(a,clk,y);

input a;

input clk;

output y;

reg y;

reg b, c;

always@(posedge clk)

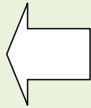
begin

    y<=c;

    c<=b;
    b<=a;

end

endmodule
    
```



Procedural 'always' block is triggered on positive edge of clock 'clk'.

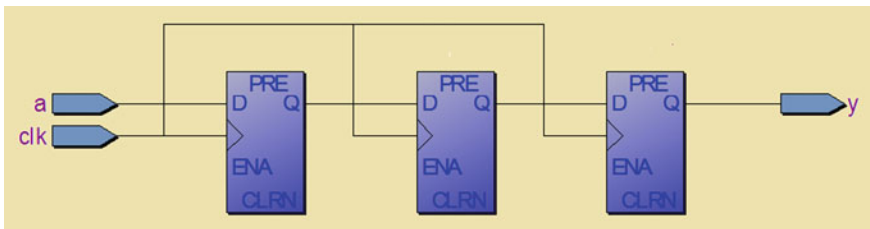
In between 'begin' and 'end' block, multiple non-blocking assignment statements are written.

Intermediate variable value 'c' is assigned to output 'y' and infers single register.

The value of 'b' is assigned to intermediate variable 'c' and an input 'a' is assigned to intermediate variable 'b'.

It infers serial input serial output shift register.

**Example 6.5** Nonblocking assignment in the same always block



**Fig. 6.5** Synthesized logic for nonblocking assignments in the same always block

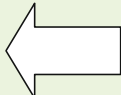
### 6.2.2 Ordering of Non-blocking Assignments

Consider the design scenario described in the example and its intention is to create the three-bit serial-input and serial-output shift register and nonblocking assignments are used.

The order of nonblocking assignments used in Sect. 6.2.2, are reordered in this Example 6.6.

Synthesized logic is shown in Fig. 6.5 and has inputs as ‘a’, ‘clk’, and an output ‘y’. The required functionality is serial-input serial-output shift register and it infers the serial-input serial-output shift register. So, the important point to remember is

```
module non_blocking_assignment(a,clk,y);  
  
input a;  
  
input clk;  
  
output y;  
  
reg y;  
  
reg b,c;  
  
always@(posedge clk)  
  
begin  
  
    b<=a;  
  
    c<=b;  
  
    y<=c;  
  
end  
endmodule
```



Procedural ‘always’ block is triggered on positive edge of clock ‘clk’.

In between ‘begin’ and ‘end’ block, multiple non-blocking assignment statements are written.

The value of ‘b’ is assigned to intermediate variable ‘c’ and an input ‘a’ is assigned to intermediate variable ‘b’.

Intermediate variable value ‘c’ is assigned to output ‘y’ and infers single register.

It infers serial input serial output shift register.

**Example 6.6** Nonblocking assignment with order change in the same always block

that, order of the nonblocking assignment statement inside the procedural “always” block is not a decisive factor in inferring logic.

## 6.3 Latch Versus Flip-Flop

In the practical sequential designs, latches and flip-flops are used as elements to design the required intended design functionality. Latch is level triggered and flip-flop is edge triggered. Most of the ASIC design uses flip-flops as sequential element.

### 6.3.1 *D Flip-Flop*

As discussed earlier, flip-flop is edge triggered and the area for the flip-flop cell is more as compared to latch and even for flip-flop additional power control logic is required as power consumption due to free running clock is higher. Flip-flop does not have the cycle stealing or time borrowing concept. The operation need to be completed in one clock cycle. For flip-flop-based design, the setup time should be met and overall operating frequency of design depends upon the critical path in the design. Timing analysis and time budgeting is more clear for flip-flop-based designs.

The D flip-flop RTL is described in Example 6.7 and uses the nonblocking assignment. Input ‘D’ is assigned to output ‘Q’ on positive edge of clock.

The synthesized logic for the positive edge triggered D flip-flop is shown in Fig. 6.6.

### 6.3.2 *Latch*

As discussed earlier, Latch is level triggered and the area for the latch cell is less as compared to flip-flop and even for the latch additional power control logic is not required as power consumption is lesser due to low switching at latch enable input. Latch has the cycle stealing or time borrowing concept and is useful in pipelining. It is not necessary that the operation need not to be completed in one clock cycle. For latch-based design, the overall operating frequency of design does not depend upon the slowest path in the design. Timing analysis and time budgeting is more difficult for latch-based designs.

```

module D_flip_flop(D,CLK,Q);

input D;

input CLK;

output Q;

reg Q;

always@(posedge CLK)

begin

    Q<=D;

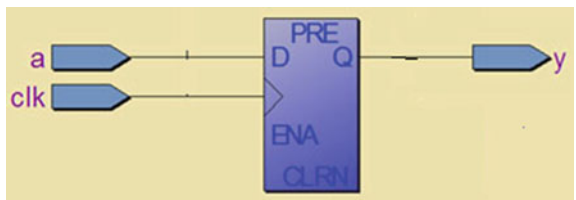
end

endmodule
    
```

On the positive edge of clock signal 'CLK' and input 'D' is assigned to output 'Q'.

It infers D flip-flop triggered on positive edge of clock.

**Example 6.7** D flip-flop using nonblocking assignment



**Fig. 6.6** Synthesized D flip-flop

The D Latch RTL is described in Example 6.8 and uses the nonblocking assignment. Input 'D' is assigned to output 'Q' on positive level of latch enable input.

The synthesized logic for positive level sensitive latch is shown in Fig. 6.7.

```

module D_latch(D,LE,Q);

input D;

input LE;

output Q;

reg Q;

always@ (D or EN)

begin


    Q <=D;

end

endmodule
    
```

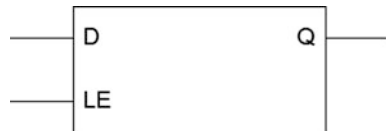
On the positive level of latch enable signal 'LE' an input 'D' is assigned to output 'Q'.

It infers D latch sensitive to positive level of latch enable input 'LE'.



**Example 6.8** Positive level sensitive D latch

**Fig. 6.7** Synthesized D latch



### 6.4 Use of Synchronous Versus Asynchronous Reset

Most of the time, the design engineer gets confused while using reset input! When to use an asynchronous reset and when to use synchronous reset is one of the key challenges for the engineer. So, for an ASIC design engineer it is required to have good understanding about the reset and rest issues as well as reset trees. Reset tree structure and synchronization will be discussed subsequently. This section describes about the synchronous and asynchronous reset description using Verilog HDL.

### **6.4.1 Asynchronous Reset D Flip-Flop**

As discussed in the Chap. 5, an asynchronous reset is an issue in ASIC design as it is independent of clock signal. The reset signal is used to initialize the sequential logic at any instance of time irrespective of clock. Reset input is not part of data path and even internally generated resets or asynchronous resets are not recommended in the ASIC design as they are prone to glitches. Even reset recovery is an issue and if asynchronous reset inputs are used then it is recommended that an asynchronous reset input can be synchronized using two-stage level synchronizer.

As shown in Example 6.9, an asynchronous reset signal D flip-flop RTL is described by using Verilog HDL.

The synthesized logic for D flip-flop with asynchronous reset is shown in Fig. 6.8.

### **6.4.2 Synchronous Reset D Flip\_Flop**

As discussed in Chap. 5, a synchronous reset is as better in ASIC design as it is dependent of clock signal. The reset signal is used to initialize the sequential logic at instance of time on the positive clock edge. Reset input is part of data path and not prone to glitches. Even reset recovery is not an issue and if synchronous reset inputs are used. Synchronous reset does not need use of level synchronizer.

As shown in Example 6.10, a synchronous reset signal D flip-flop RTL is described by using Verilog HDL.

The synthesized logic is shown in Fig. 6.9 where reset input is part of data path.

## **6.5 Use of If-Else Versus Case Statements**

For the sequential designs, use the “if-else” construct to describe the priority logic functionality. To assign the priority signals use the “if-else” construct. Use the “case” construct to describe the parallel logic. Please refer Chap. 5 for the detailed information about the use of “if-else” and “case” construct.

## **6.6 Internally Generated Clocks**

Internally generated clock signals use system or master clock and generates an output as internally generated clock signal. But, internally generated clock signals need to be avoided as it causes the functional and timing issues in the design. The functional and timing problems are due to the combinational logic propagation delays. The internal generated clock signals can generate the glitch or spike in the

```

module flip_flop_sync_reset(D,CLK,reset_n, Q);

input D;

input CLK;

input reset_n;

output Q;

reg Q;
always@(posedge CLK or negedge reset_n)

begin

    if(!reset_n)

        Q<=1'b0;

    else

        Q<=D;

end

endmodule

```

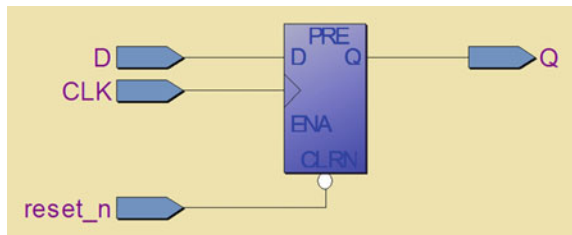
As described in the sensitivity list of 'always' block 'reset\_n' is part of the sensitivity list.

So the procedural 'always' block is invoked for changes on the clock 'CLK' or changes on 'reset\_n'.

Synthesized logic is D flip-flop with asynchronous reset.

**Example 6.9** Verilog RTL for D flip-flop with asynchronous reset

**Fig. 6.8** Synthesized D flip-flop with asynchronous reset



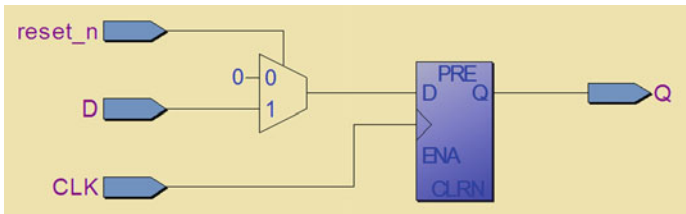
```
module flip_flop_sync_reset(D,CLK,reset_n, Q);  
  
input D;  
  
input CLK;  
input reset_n;  
  
output Q;  
  
reg Q;  
  
always@(posedge CLK) ←  
begin  
    if(!reset_n)  
        Q<=1'b0;  
  
    else  
        Q<=D;  
  
end  
  
endmodule
```

As described in the sensitivity list of 'always' block 'reset\_n' is not specified.

So the procedural 'always' block is invoked for changes on the clock 'CLK'.

Synthesized logic is D flip-flop with synchronous reset.

**Example 6.10** Verilog RTL for D flip-flop with synchronous reset



**Fig. 6.9** Synthesized D flip-flop with synchronous reset



output. This can trigger the sequential logic multiple times or can generate undesired output. Even due to violation of setup or hold time these type of designs have the timing violations.

It is always recommended to generate the internal clocks by using register output logic. But still due to the propagation delay of the flip-flop, the overall cumulative delay or skew can generate the glitches or spikes in the design.

As shown in Example 6.11, Verilog RTL is described to generate the internal clocks. The generated internal clock signal is used by some other sequential procedural block.

```

module internal_clock (in_1, in_2, clk, out_1);

input in_1, in_2, clk;

output out_1;

reg int_clk, out_1;

always @ (posedge clk)

begin

int_clk<= in_1;

end

always @ (posedge int_clk)

begin

out_1 <= in_2;

end

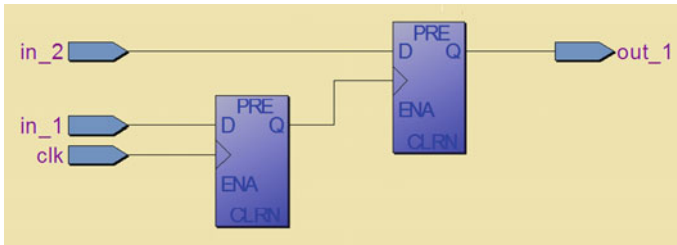
endmodule

```

The internal generated clock signal 'int\_clk' is described by using first procedural 'always' block.

Second procedural 'always' block is triggered by changes on 'int\_clk' and generates an output 'out\_1'.

**Example 6.11** Verilog RTL for internally generated clock



**Fig. 6.10** Synthesized internally generated clock logic

The synthesized logic is shown in Fig. 6.10 and the first register is driven by clock 'clk' and the second register clock is driven by 'int\_clk'.

## 6.7 Gated Clocks

Gated clock signals are used to enable switching at the clock input and can be used in single or the multiple clock domain designs by using the enable inputs. When enable input is high the clock domain is on and when enable input is low the clock domain is off. The clock gating logic is required to control the clock turn on or turn off. Clock gating is an efficient technique used in ASIC design to reduce the switching power at the clock input of register. By using the clock gating structure, the clock switching can be stopped as and when required according to the design functional requirements.

But the issue with the clock gating is it cannot be used in the synchronous designs the reason being it introduces significant amount of clock skew and even this technique introduces glitches. To avoid the glitches, special care need to be taken by ASIC design engineer.

Verilog RTL is described in Example 6.12 and uses enable input to control the clock switching activity. For 'enable=1,' the clock input 'clk' toggles and for 'enable=0' clock input is permanently active low so no switching at clock input.

The synthesized logic is shown in Fig. 6.11 where clock is gated by using AND logic.

## 6.8 Use of Pipelining in Design

Pipelining is one of the powerful techniques used to improve the performance of the design at the cost of latency. This technique is used in many processor designs and many ASIC design applications to perform multiple tasks at a time that is for concurrent execution. This section discusses about the design without pipelining and design with pipelining.

```

module clk_gating (in_1, enable, clk, out_1);

input in_1, clk, enable;

output out_1;
wire clk_gated;

reg out_1;

assign clk_gated = clk && enable;

always @ (posedge clk_gated)
begin
    out_1 <= in_1;
end

endmodule

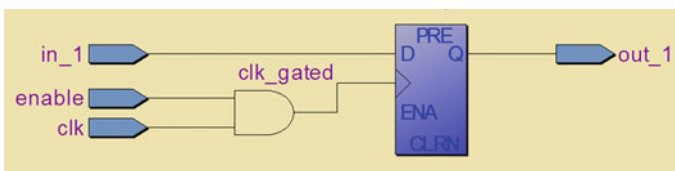
```

The source clock signal is defined as 'clk' and gated clock signal is defined as 'clk\_gated'.

The 'clk\_gated' signal is generated by using continuous assignment statement 'assign' and it is logical AND of 'clk' and 'enable' input.

The 'clk\_gated' signal is used to invoke the 'always' block and hence it infers the D flip-flop with the clock gating logic at the clock input of flip-flop.

**Example 6.12** Verilog RTL for clock gating



**Fig. 6.11** Synthesized clock gating logic

### 6.8.1 Design Without Pipelining

During the initial stage of the design, most of the designs are described by using Verilog RTL without the use of the pipelined logic. If the desired speed that is design performance is not met, then ASIC designer can tweak the design by using

various approaches. One of the best approaches is pipelining by inserting the register logic according to the clock latency and data rate requirements.

Example 6.13 describes the design functionality using Verilog RTL without the use of any pipelined logic.

The synthesized RTL logic is described in Fig. 6.12 and consists of two registers triggered by common clock source 'clk'.

### 6.8.2 Design with Pipelining

To improve the design performance, the combinational logic AND output can be given to the additional pipelined register and the output of the pipelined register can drive one of the input of OR logic.

This technique will improve the overall performance of the design at the cost of once clock latency. The improvement in the design performance is due to the reduction in the combinational delay in the register-to-register path.

Verilog RTL is described in Example 6.14, and by adding additional register logic the pipelined is achieved.

The synthesized RTL logic is described in Fig. 6.13 and consists of three registers triggered by common clock source 'clk'.

## 6.9 Guidelines for Modeling Synchronous Designs

Following are key guidelines used to describe the synchronous designs

1. To describe the functionality of synchronous designs, use the nonblocking assignments.
2. Do not use the latch-based designs as latches are transparent for half a clock cycle.
3. Use the pipelined stages to improve the design performance.
4. Use the synchronous reset signals as they are not prone to glitches or spikes.
5. If asynchronous signals are used then use the dual stage synchronizers to synchronize the internally generated resets.
6. Use clock gating cells for low-power design.

## 6.10 Multiple Clocks in the Same Module

Multiple clock sources or signals can be used in the multiple clock domain designs. These clock signals can be generated by using different sources and can be used in the ASIC design to trigger the different "always" procedural blocks. The data

```

module design_without_pipeline ( a_in, b_in, clk, reset_n, q_out);

input a_in;

input b_in;

input clk;

input reset_n;

output reg q_out;

reg q1_out;

always @ (posedge clk or negedge reset_n)

begin
    if (~reset_n)
        begin
            { q_out, q1_out } <= 2'b00;
        end
    else
        begin
            q1_out <= a_in;
            q_out <= ( (q1_out & a_in ) | b_in );
        end
    end
end

endmodule

```

The 'always' procedural block is sensitive to changes on the clock 'clk' and reset 'reset\_n'.

This block is used to infer the sequential logic triggered on positive edge of clock 'clk'. The reset is described as an asynchronous active low reset 'reset\_n'.

Then functionality is described using Verilog RTL to infer two registers with the combinational logic.

The structure described is RTL without any pipelined logic.

**Example 6.13** Verilog RTL without pipelined stage

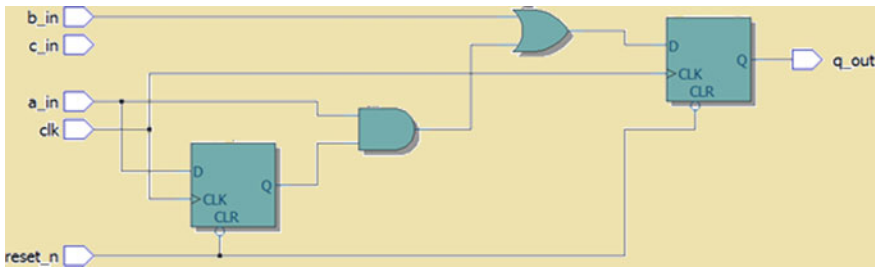


Fig. 6.12 Synthesized Logic without pipelined stage

transfer from one clock domain to another clock domain needs additional synchronizers in the data path and control path and these can be discussed in the subsequent chapters.

Verilog RTL is described in Example 6.15 and uses two different clock signals ‘clk1’ and ‘clk2’. Two different procedural blocks are used to describe the functionality y using ‘clk1’ and ‘clk2’, respectively. Defining the multiple clock signals in the same module is not the best practice. In the multiple clock domain designs, according to the functional requirements the different design blocks need to be described and they can be triggered on different clock signals.

Synthesized output is shown in Fig. 6.14 and generates an output ‘f1\_out’ and ‘f2\_out’. The clock signal ‘clk1’ is used to trigger the upper register. Upper register is triggered on positive edge of clock ‘clk1’. The lower register is triggered on negative edge of ‘clk2’.

### 6.11 Multi Phase Clocks in the Design

The clock signals used to trigger multiple procedural blocks and generated from the same clock source and having the arrival time difference are called as multi phase clock signals. For example, if one of the procedural blocks is triggered on positive edge of clock and another procedural block is triggered on negative edge of clock then there is phase difference of 180° in the triggering of the register and these signals and treated like phase shifted signals.

Verilog RTL is shown in Example 6.15 and one of the procedural block is triggered by positive edge of clock and another is triggered by negative edge of clock (Example 6.16).

The synthesized logic is shown in Fig. 6.15 where two different registers are triggered on different edges of clocks.

```

module design_without_pipeline ( a_in, b_in, clk, reset_n, q_out);

input a_in;

input b_in;

input clk;

input reset_n;

output reg q_out;

reg q1_out, q2_out;
always @(posedge clk or negedge reset_n)

begin

    if (~reset_n)

        begin

            { q_out, q2_out, q1_out} <= 3'b000;

        end

    else

        begin

            q1_out <= a_in;

            q2_out <= (q1_out & a_in )

            q_out <= (q2_out | b_in);

        end

    end

end

endmodule

```

The 'always' procedural block is sensitive to changes on the clock 'clk' and reset 'reset\_n'.

This block is used to infer the sequential logic triggered on positive edge of clock 'clk'. The reset is described as an asynchronous active low reset 'reset\_n'.

The functionality is described using Verilog RTL to infer two registers with the combinational logic. But to improve the speed of the design the combinational logic is spitted using additional register.

The structure described is RTL with single stage pipelined logic.

**Example 6.14** Verilog RTL with pipelined stage

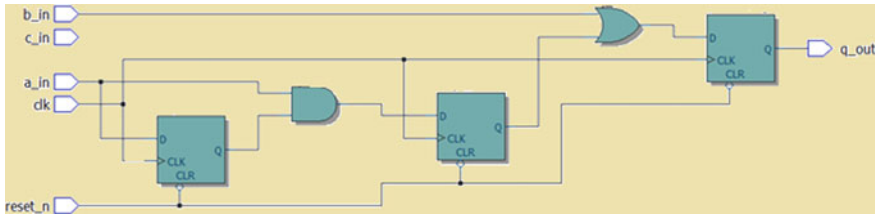


Fig. 6.13 Synthesized logic with pipelined stage

```

module multi_clock_gen (clk1, clk2, a_in, b_in, c_in, f1_out,
    f2_out);
input clk1;
input clk2;
input a_in;
input b_in;
input c_in;
output reg f1_out;
output reg f2_out;
always @ (posedge clk1)
f1_out <= a_in & b_in;
always @ (negedge clk2)
f2_out <= b_in ^ c_in;
endmodule
    
```

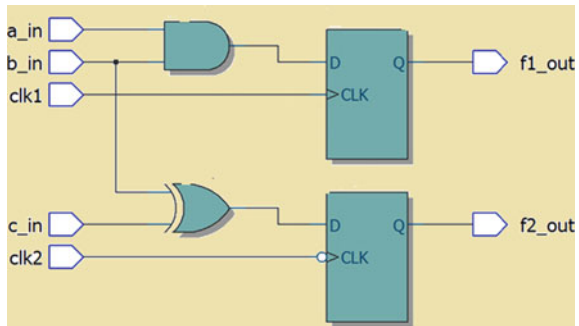
Multiple clocks can be defined in the same module and used by two different procedural 'always' blocks.

Multiple clocks are used in multiple clock domain designs.



Example 6.15 Verilog RTL for multiple clock definitions

Fig. 6.14 Synthesized logic for multiple clock





```

module multi_phase_clk (a_in, b_in, clk , f_out);

input a_in;

input b_in;

input clk;

output reg f_out;

reg t_out;
always @ (posedge clk)

f_out <= t_out & b_in;

always @ (negedge clk)

t_out <= a_in | b_in;

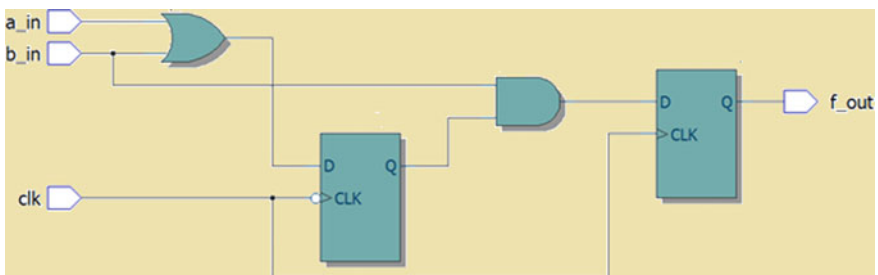
endmodule

```

Multiphase clock signals can be used in few applications to generate the different outputs on the positive and negative edge of clock.

In the Verilog RTL one of the 'always' block is triggered on positive edge of clock and another 'always' block is triggered on the negative edge of clock.

**Example 6.16** Verilog RTL with multiphase clock



**Fig. 6.15** Synthesized logic for the multi phase clock

## 6.12 Guidelines for Modeling Asynchronous Designs

Following are key guidelines needed to be followed while modeling the asynchronous design

1. If asynchronous reset signals are used, then use the dual edge synchronizer to synchronize the internally generated reset signals.
2. Avoid use of driving the flip-flop output to the asynchronous reset of the subsequent flip-flop as this can have the race conditions.
3. Avoid use of asynchronous pulse generator as it creates the issue in the design and timing closure and even during the place and route.
4. If power consumption is the goal then only use the efficient ripple counter; but there is performance degrade while using the ripple counters due to the cumulative delay effect or the cascaded delay due to the individual propagation delay of flip-flop.

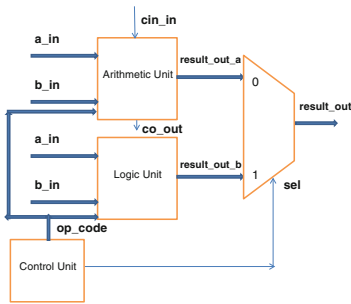
## 6.13 Summary

To summarize this chapter, the following are key highlights

1. Do not use blocking assignments while describing the sequential designs.
2. Use the flip-flop-based designs instead of latch-based design.
3. To reduce the overall switching power consumption, use the low power clock gating cells.
4. Use the synchronous resets in the design and if asynchronous reset is used then use the reset synchronizers.
5. Use pipelining to improve the design performance.

# Chapter 7

## Complex Designs Using Verilog RTL



The complex designs can be efficiently implemented by using the Verilog. Now days design complexity has increased and the designs are targeted for the lower power, high speed and least area. This chapter discusses the use of Verilog constructs to design logic for the required functionality.

**Abstract** The complex ASIC designs can be described by using the Verilog RTL. In the practical scenario the objective is to describe the design functionality by using efficient Verilog RTL by using key and important combinational and sequential design guidelines. This chapter focuses on the discussion to describe the complex designs like ALU, parity generators, checkers, and barrel shifters. This chapter also discusses about the synthesized logic with the data path and control paths. The complex examples are explained in this chapter with practical aspects and with the diagrams and functional tables. This chapter is useful for ASIC and FPGA designers to understand the issues like combinational designs, critical paths, register inputs, and outputs.

**Keywords** ALU · Logic unit · Arithmetic unit · Data path · Control path · Function · Task · Timing control · Delay control · Parity checker · Parity generator · Combinational shifter · Protocol · Register input · Register output · Barrel shifter · DSP

As discussed in the previous chapters Verilog HDL is efficiently used to code the functionality of the design. The concurrent and sequential constructs discussed in the previous chapters can be used to infer the synthesized logic. In the practical

ASIC designs, the design functionality is complex and needs to be described by using the synthesizable Verilog RTL to infer the gate level netlist with optimal design performance. Most of the ASIC and SOCs uses the processors, busses, arbiters and protocols (predefined set of rules or transactions). An efficient Verilog coding is a very important aspect while describing the functionality of above blocks. In such scenarios ASIC designer needs to use the synthesizable constructs with combinational and sequential design guidelines.

The subsequent section discusses about the efficient complex designs and practical scenario while describing the processor computational logic, barrel shifters, parity generators, checkers, and tasks and functions.

## 7.1 ALU Design

Arithmetic logic unit (ALU) is used in most of the processors to perform the arithmetic and logical operations. Processor performs one of the operations at a time depending on the operational code (opcode). For 8-bit processors, the ALU is used to perform the operations on two eight-bit operands. Operand is the data on which operation needs to be performed. Similarly for the 16-bit processors the ALU is used to perform the operations on two 16-bit numbers.

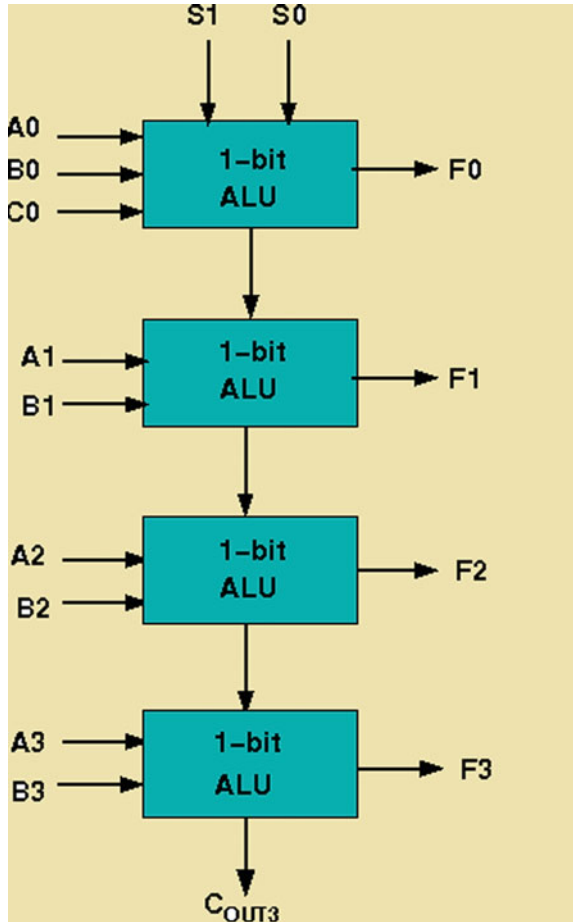
As shown in Fig. 7.1 a ALU architecture is described to perform the operation on two four-bit numbers A (A3 is MSB and A0 is LSB), B (B3 is MSB and B0 is LSB), and carry input C<sub>0</sub>, A ALU generates an output F (F3 is MSB and F0 is LSB) and an output carries C<sub>out3</sub>. In the practical ASIC design scenario, one-bit ALU is designed to perform operation on the single bit of data. The operation is performed depending on the opcode bits specified by lines S1, S0. As shown in figure, ALU is designed to perform the execution for the four instructions. Table 7.1 for ALU and the functionality is described to perform the operational listed depending on the status of select lines 'S1' and 'S0'. In this example opcode is 2-bit and is indicated by 'S1' and 'S0'.

### 7.1.1 Logical Unit Design

In the practical ASIC design scenario, it is recommended to describe the functionality of design using an efficient Verilog RTL. So at the microarchitecture level the design is partitioned into multiple modules. The partitioning of design gives the better design understanding and visibility to designer. Consider a scenario to implement the design functionality of an 8-bit ALU, the design is petitioned as separate logic unit and arithmetic unit. Separate arithmetic and logical unit functionality can be described by using efficient Verilog RTL for better readability and better synthesis outcome.

Figure 7.2 is shown below and used to implement the four logical operations and these logical operations are described in the functional table. The logic unit is

**Fig. 7.1** Four-bit ALU architecture



**Table 7.1** Four bit ALU operational table

S1	S0	Operation
0	0	Addition of A, B without carry
0	1	Subtraction of A, B without borrow
1	0	XOR of A, B
1	1	Complement of A

performing either AND, OR, XOR or complement operation. Table 7.2 shown below describes the different logical operations. The complement operation is performed by using adder having one input  $A_0$  and another input logical '1'.

The issue with this type of design is due to the use of parallel and multiplexing logic. The data path is described from input  $A_0$  and  $B_0$  to the multiplexer data inputs and control path is the control lines of multiplexers 'S1' and 'S0'. As shown in Fig. 7.2 the logic unit performs all the operations at a time and result 'F<sub>0</sub>' for one of the operation results. But this technique is inefficient as it needs more area, power

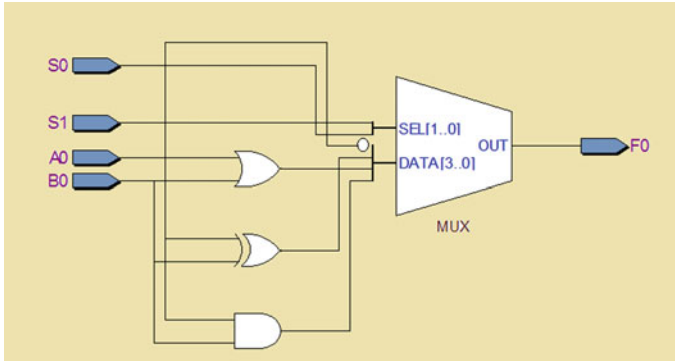


Fig. 7.2 Single-bit logic unit

Table 7.2 Single-bit logical unit operational table

S1	S0	Operation
0	0	A0 AND B0
0	1	A0 OR B0
1	0	XOR of A0, B0
1	1	Complement of A0

and it does not have the efficient implementation mechanism. If ‘S1’, ‘S0’ are late arriving signals and if this block is used in the register-to-register path then there may be possibility of the timing violations. Another important aspect is the concept of resource sharing which is not used in this design.

So, it is recommended to write an efficient Verilog RTL for the logic unit using the “case” construct but by sharing the common resources. The following section describes the Verilog RTL for the logic unit to infer the parallel logic and the logic with the registered inputs and outputs.

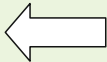
### 7.1.1.1 Logic Unit to Infer Parallel Logic

Example 7.1 describes the functionality to perform the operations on two 8-bit binary inputs “a\_in”, “b\_in”. The operations are performed using the functionality which is described in Table 7.3. The Verilog RTL infers the parallel logic with multiplier encoding.

As described in Example 7.1 the functionality is described by using a procedural “always” block with the “case” construct. All the case conditions are described and during “default” condition the logic unit generates output “result\_out” equal to ‘8'b0000\_0000’.

The functionality of the logic unit can be modeled using the full-case construct. As described in Example 7.2 the functionality is described by using a procedural “always” block with the full “case” construct. All the case conditions are described using the full-case construct.

```
module logical_unit ( a_in, b_in, op_code, result_out);  
  
input [7:0] a_in;  
  
input [7:0] b_in;  
  
input [1:0] op_code;  
  
output reg [7:0] result_out;  
  
always@ ( a_in, b_in, op_code)  
  
begin  
  
case ( op_code)  
2'b00 : result_out = a_in | b_in;  
  
2'b01 : result_out = a_in ^ b_in;  
  
2'b10 : result_out = a_in & b_in;  
  
2'b11 : result_out = ! a_in;  
  
default : result_out = 8'b0000_0000;  
  
endcase  
  
end  
  
endmodule
```



The procedural block is level sensitive to changes on 'a\_in', 'b\_in' and 'op\_code'.

'case' construct is used to infer the parallel logic for the required logical operations.

Please refer the functional table for the logical operations.

'default' condition is specified and the result during 'default' condition is equal to '0000\_0000'.

**Example 7.1** Verilog RTL for 8-bit logic unit

Synthesized logic using full case construct for the 8-bit logic unit is shown in Fig. 7.3. As shown in the above figure it infers the logic gates with multiplexing logic. In the practical scenario it is recommended to use the adders as common resources to implement both the logic and arithmetic unit. Readers are requested to refer Chap. 9 for the improved design of 8-bit ALU.

**Table 7.3** Operational Table for 8-bit ALU

op_code[1]	op_code[0]	Logic operation
0	0	a_in OR b_in
0	1	a_in XOR b_in
1	0	a_in AND b_in
1	1	Complement of a_in

```

module logical_unit ( a_in, b_in, op_code, result_out);
input [3:0] a_in;

input [3:0] b_in;

input [1:0] op_code;

output reg [3:0] result_out;

always@ ( a_in, b_in, op_code)

begin

case ( op_code)

2'b00 : result_out = a_in | b_in;

2'b01 : result_out = a_in ^ b_in;

2'b10 : result_out = a_in & b_in;

2'b11 : result_out = ! a_in;

endcase

end

endmodule

```



The procedural block is level sensitive to changes on 'a\_in', 'b\_in' and 'op\_code'.

'case' construct is used to infer the parallel logic for the required logical operations.

Please refer the functional table for the logical operations.

All 'case' conditions are covered and hence the 'case' construct is full case. It infers the parallel logic without latches.

**Example 7.2** Verilog RTL for 8-bit ALU using full-case construct



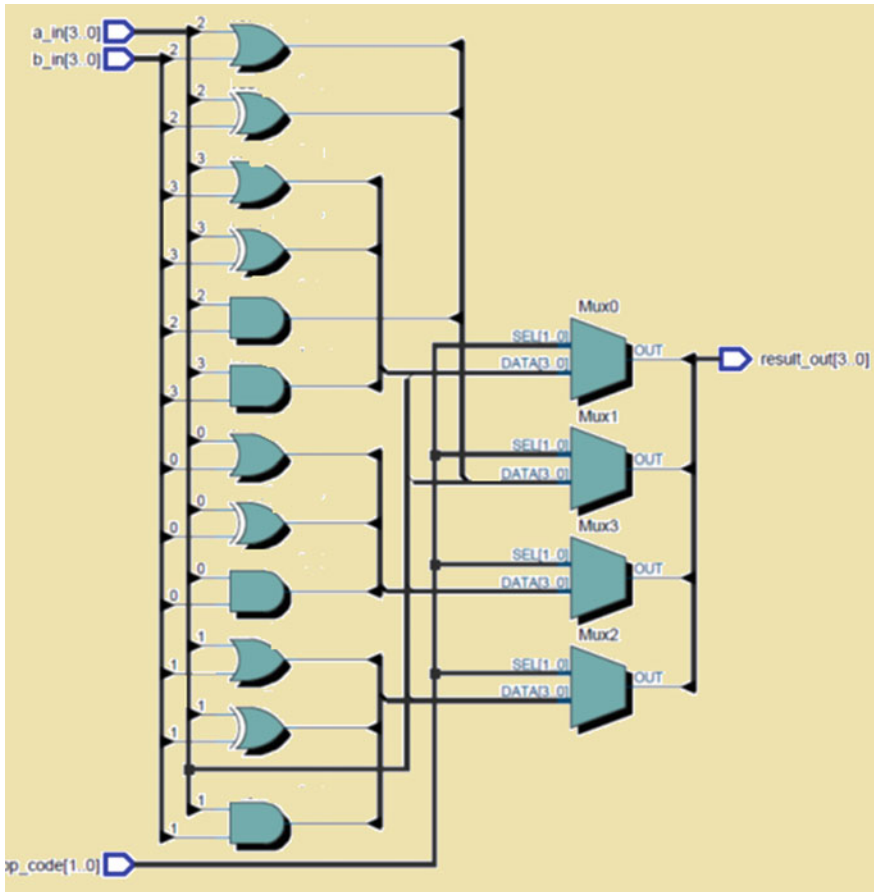


Fig. 7.3 Synthesized logic for 8-bit logic unit

### 7.1.1.2 Logical Unit with Registered Inputs and Outputs

For the efficient and clean timing analysis it is recommended to use register inputs and register outputs. If all the inputs are registered that is data sampled on the active edge of clock and even if all the outputs are registered and captured on the active edge of clock then design can give better understanding about the register-to-register timing paths. The registered inputs and registered outputs can give the clean data path and even the output is glitch or hazard free. For the performance improvement the pipelining can be used to reduce the data arrival time. Please refer the Chap. 11 for the detail information about the timing analysis.

Example 7.3 uses the register input and register output logic. The inputs are sampled or captured on the positive edge of clock “clk” and outputs are launched at

```

module logical_unit_register_io( a_in, b_in, op_code, clk, reset_n, re-
    sult_out);

input [3:0] a_in;
input [3:0] b_in;
input [1:0] op_code;
input clk;
input reset_n;
output reg [3:0] result_out;
reg [3:0] reg_a_in;
reg [3:0] reg_b_in;
reg [1:0] reg_op_code;
reg [3:0] reg_result_out;

always @ ( posedge clk or negedge reset_n)
begin
    if ( ~reset_n)
        { reg_a_in, reg_b_in, reg_op_code} <= 3'b0;
    else
        { reg_a_in, reg_b_in, reg_op_code} <= { a_in, b_in, op_code};

end

always@(reg_a_in, reg_b_in, reg_op_code)
begin
    case ( reg_op_code)
2'b00 : reg_result_out = reg_a_in | reg_b_in;
2'b01 : reg_result_out = reg_a_in ^ reg_b_in;
2'b10 : reg_result_out = reg_a_in & reg_b_in;
default : reg_result_out = !reg_a_in;
    endcase
end

always @ (posedge clk or negedge reset_n)
begin
    if (~reset_n)
        result_out <= 4'b0000;
    else
        result_out <= reg_result_out;
    end
endmodule

```

For the better and clean timing analysis it is recommended to use the register inputs and register outputs.

The procedural block is triggered on positive edge of clock and used to sample the data inputs 'a\_in', 'b\_in', 'opcode'.

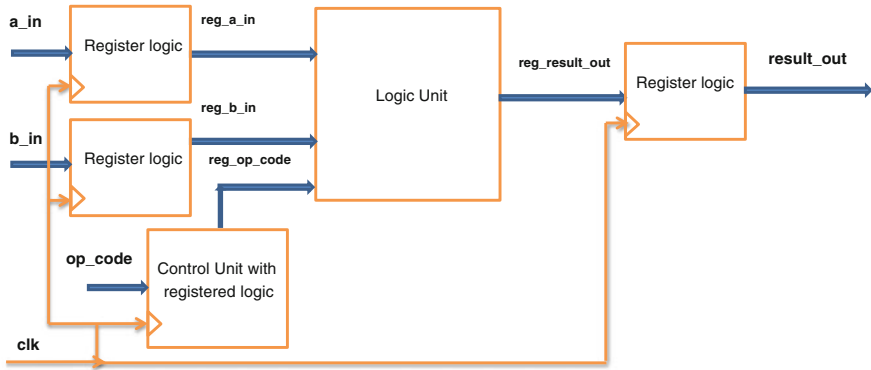
The procedural block is level sensitive to changes on registered inputs 'reg\_a\_in', 'reg\_b\_in' and 'reg\_op\_code'.

'case' construct is used to infer the parallel logic for the required logical operations.

Please refer the functional table for the logical operations.

2'b11 or 'default' condition generates output which is equal to complement of registered input 'reg\_a\_in'.

**Example 7.3** Verilog RTL for 8-bit logic unit with register inputs and outputs



**Fig. 7.4** Synthesized logic unit with registered inputs and outputs

the positive edge of “clk”. During reset condition ‘reset\_n=0’ the logical unit is initialized to logic ‘0’.

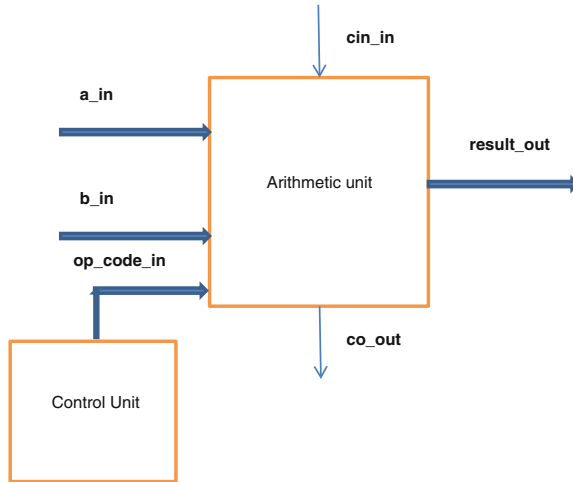
The above example generates the logic unit with all the inputs and outputs registered on positive edge of clock. Readers are requested to assume that every register has an asynchronous reset input “reset\_n”. The synthesized logic is shown in Fig. 7.4.

### 7.1.2 Arithmetic Unit

Arithmetic unit is used to perform the arithmetic operations such as addition, subtraction, increment, and decrement. The operations are performed on the two different operands. The functional Table 7.4 gives information about the different operations need to be performed. Arithmetic unit is described in such a way that it performs only one operation at time. Figure 7.5 describes the signal connectivity of different input and output signals (Example 7.4).

**Table 7.4** Operational table for the arithmetic unit

op_code[2]	op_code[1]	op_code[0]	Logic operation
0	0	0	Transfer a_in
0	0	1	a_in ADD b_in
0	1	0	a_in ADD b_in with carry input cin_in
0	1	1	a_in SUB b_in
1	0	0	a_in SUB b_in with borrow input cin_in
1	0	1	Increment a_in
1	1	0	Decrement b_in
1	1	1	No operation performed



**Fig. 7.5** Block diagram of arithmetic unit

```

module arithmetic_unit ( a_in, b_in, cin_in, op_code_in, result_out,
    co_out);

input [3:0] a_in;
input [3:0] b_in;
input cin_in;
input [2:0] op_code_in;
output reg [3:0] result_out;
output reg co_out;

always @ ( a_in, b_in, cin_in, op_code_in)
begin

    case ( op_code_in)
        3'b000 : {co_out,result_out} = {1'b0,a_in} ;
        3'b001 : {co_out,result_out} = a_in + b_in ;
        3'b010 : {co_out,result_out} = a_in + b_in + cin_in ;
        3'b011 : {co_out,result_out} = a_in - b_in ;
        3'b100 : {co_out,result_out} = a_in - b_in - cin_in;
        3'b101 : {co_out,result_out} = a_in + 1'b1 ;
        3'b110 : {co_out,result_out} = a_in - 1'b1 ;
        default : {co_out,result_out} = 9'b0_0000_0000 ;
    endcase

end

endmodule
  
```

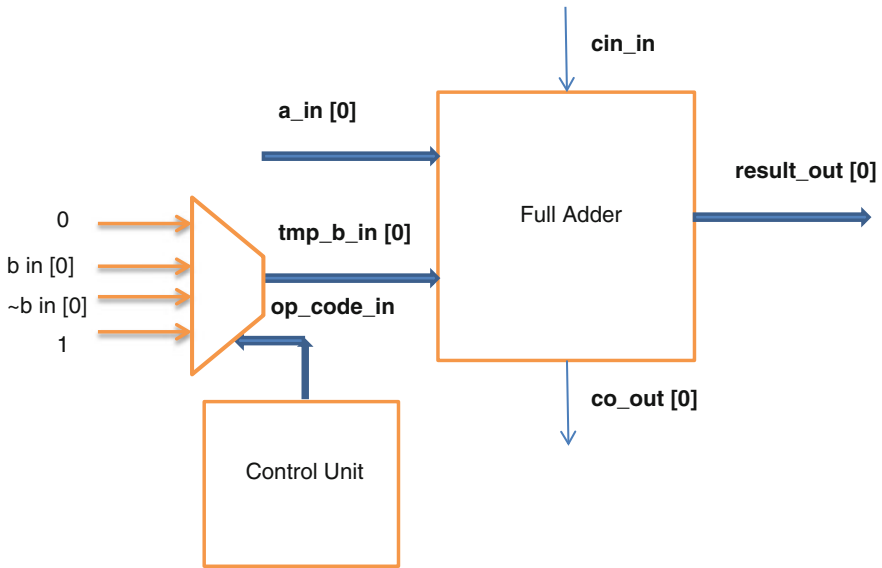
The procedural 'always' block is used and is level sensitive to changes on 'a\_in', 'b\_in', 'cin\_in' and an 'op\_code\_in'.

The functionality is described by using the 'case' construct.

'case' construct is used to infer the parallel logic and in generates the synthesized result without latches due to 'default' clause.

Please refer the operational table for the arithmetic unit.

**Example 7.4** Verilog RTL for the arithmetic unit



**Fig. 7.6** Synthesized one bit Arithmetic unit

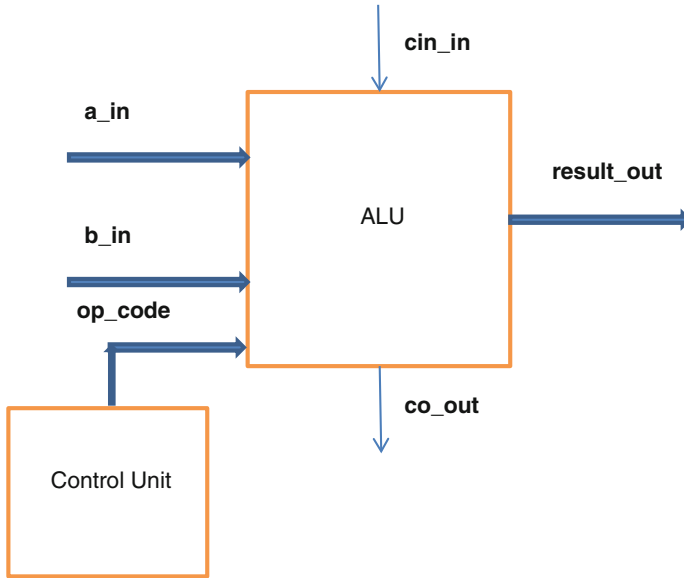
The synthesized logic for one-bit arithmetic unit is shown in Fig. 7.6. The logic uses the full adder as component to perform the addition and subtraction. Subtraction is performed using 2’s complement addition. The synthesized logic also consists of the multiplexer 4:1 to pass the required operand at one of the input of full adder depending on the opcode.

### 7.1.3 Arithmetic and Logical Unit

Figure 7.7 illustrates the ALU with the associated logic circuit to perform the operation on two 8-bit numbers “a\_in”, “b\_in”. For logic operations, the carry input (cin\_in) is ignored and the output “result\_out” is generated depending on the operational code of the instruction. Depending on the operational code ALU can perform either arithmetic or logical operation. During arithmetic operations if result is more than 8-bit then carry output “co\_out” is set to logical ‘1’ that indicates carry propagation outside to MSB (Table 7.5).

Table 7.6 describes the number of bits required at inputs and outputs for the ALU design for 11 instructions. The table describes the seven arithmetic instructions and four logical instructions. The pin or signal description is shown in Table 7.5.

An efficient Verilog RTL description using the two different “case” constructs to infer the parallel logic is described in Example 7.5. For the ‘op\_code\_in[3]=0’



**Fig. 7.7** ALU top level diagram

**Table 7.5** Signal or pin description of 8-bit ALU

Signal or pin name	Size (bits)	Description
a_in	8	An 8-bit operand
b_in	8	An 8-bit operand
cin_in	1	Carry input to a ALU
op_code_in	4	4-bit opcode for instruction
result_out	8	An 8-bit output from ALU
co_out	1	One bit output carry from ALU

**Table 7.6** Operational table for 8-bit ALU

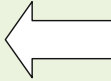
Operational code	Instruction	Description
0000	Transfer a_in	Generate an output $a\_in + 0 + 0$
0001	Addition without carry	$a\_in + b\_in + 0$
0010	Addition with carry	$a\_in + b\_in + 1$
0011	Subtract without borrow	$a\_in - b\_in$
0100	Subtract with borrow	$a\_in - b\_in - 1$
0101	Increment a_in by 1	$a\_in + 1$
0110	Decrement a_in by 1	$a\_in - 1$
1000	a_in OR with b_in	$a\_in \text{ OR } b\_in$
1001	a_in XOR with b_in	$a\_in \text{ XOR } b\_in$
1010	a_in AND with b_in	$a\_in \text{ AND } b\_in$
1011	Complement a_in	Not a_in

```

module arithmetic_logic_unit ( a_in, b_in, cin_in, op_code_in, re-
    sult_out, co_out);

input [3:0] a_in;
input [3:0] b_in;
input cin_in;
input [3:0] op_code_in;
output reg [3:0] result_out;
output reg co_out;

always @ ( a_in, b_in, cin_in, op_code_in)
begin
    if (~op_code_in[3])
        begin
            case (op_code_in[2:0])
                3'b000 : {co_out,result_out} = {1'b0,a_in} ;
                3'b001 : {co_out,result_out} = a_in + b_in;
                3'b010 : {co_out,result_out} = a_in + b_in + cin_in ;
                3'b011 : {co_out,result_out} = a_in - b_in ;
                3'b100 : {co_out,result_out} = a_in - b_in - cin_in;
                3'b101 : {co_out,result_out} = a_in + 1'b1;
                3'b110 : {co_out,result_out} = a_in - 1'b1;
                default : {co_out,result_out} = 9'b0_0000_0000 ;
            endcase
        end
    else
        begin
            case ( op_code_in [2:0])
                3'b000 : {co_out,result_out} = {1'b0, (a_in | b_in) };
                3'b001 : {co_out,result_out} = {1'b0, (a_in ^ b_in) };
                3'b010 : {co_out,result_out} = { 1'b0, (a_in & b_in) };
                3'b011 : {co_out,result_out} = { 1'b0, ~a_in };
                default : {co_out,result_out} = 9'b0_0000_0000 ;
            endcase
        end
    end
endmodule
    
```



The procedural 'always' block is used and is level sensitive to changes on 'a\_in', 'b\_in', 'cin\_in' and an 'op\_code\_in'.

The functionality is described by using the 'case' construct.

'case' construct is used to infer the parallel logic and in generates the synthesized result without latches due to 'default' clause.

Please refer the operational table for the arithmetic logic unit.



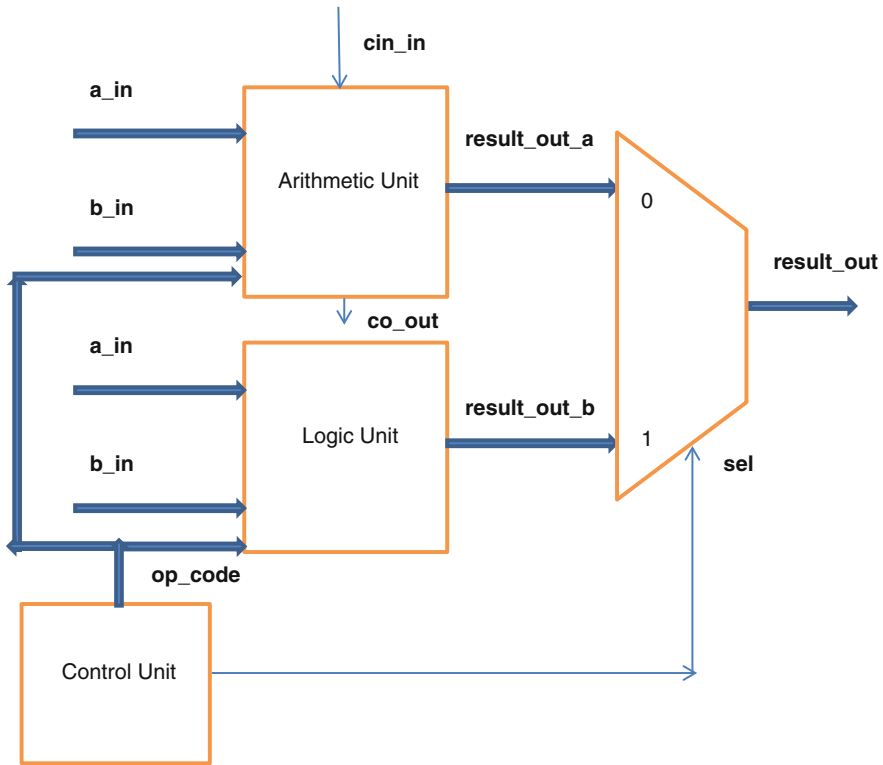
The procedural block is level sensitive to changes on 'a\_in', 'b\_in' and 'op\_code\_in'.

'case' construct is used to infer the parallel logic for the required logical operations.

Please refer the functional table for the logical operations.

All 'case' conditions are covered and hence the 'case' construct is full case. It infers the parallel logic without latches.

**Example 7.5** Verilog RTL for 8-bit ALU



**Fig. 7.8** Synthesized 8-bit ALU

it performs the arithmetic operation and when ‘`op_code_in[3]=1`’ it performs the logic operation.

The synthesized diagram for the 8-bit ALU is shown in Fig. 7.8. As shown in the figure it consists of the parallel logic for the arithmetic operations and logic operations. Using the multiplexer at the output side either arithmetic or logical operation result is generated. The logic does not use the concept of resource sharing and area, power optimization.

## 7.2 Functions and Tasks

Task and functions are used in the Verilog to describe the commonly used functional behavior. Instead of replicating the same code at the different places, it is good and common practice to use the functions or tasks depending on the requirement. For easy maintenance of the code, it is better to use the functions or tasks like the subroutine.



### 7.2.1 *Counting 1's from the Given String*

The following example describes the task used to count 1's from the given string. The following are key important points need to remember while using the task:

1. Task can consist of the time control statements and even delay operators.
2. Task can have input and output declarations.
3. Task can consist of function calls but function cannot consist of the task.
4. Task can have output argument and not used to return the value when called.
5. Task can be used to call other tasks.
6. It is not recommended to use the task while writing the synthesizable Verilog RTL.
7. Tasks are used for writing the behavioral or simulatable model.

Example 7.6 is the description to count number of 1's from the given string. In this example task is used with arguments “data\_in”, “out” The name of task is “count\_1s\_in\_byte”. In most of the protocol descriptions it is required to perform some operations on the input string. In this example the string is 8-bit input “data\_in” and output result is 4-bit “out”. It is not recommended to use the task to generate synthesized logic.

### 7.2.2 *Module to Count 1's using Functions*

The following example describes the function used to count 1's from the given string. The following are the key important points need to remember while using the function:

1. Function cannot consists of the time control statements and even delay operators.
2. Function can have at least one input argument declarations.
3. Function can consist of function calls but function cannot consists of the task.
4. Function executes in zero simulation time and returns single value when called.
5. It is not recommended to use the function while writing the synthesizable Verilog RTL.
6. Functions are used for writing the behavioral or simulatable model.
7. Functions should not have nonblocking assignments.

Example 7.7 is the description to count number of 1's from the given string. In this example, function is used with arguments “data\_in”. The name of function is “count\_1s\_in\_byte”. In most of the protocol descriptions, it is required to perform some operations on the input string. In this example the string is 8-bit input “data\_in” and output result is 4-bit “out”. It is not recommended to use the function to generate synthesized logic.

```

module count_one (data_in, out);

input [7:0] data_in;

output reg [3:0] out;

always @(data_in)

count_1s_in_byte(data_in, out);

// task declaration from here

task count_1s_in_byte(input [7:0] data_in, output reg [3:0] count);

integer i;

begin // task functional description

count = 0;

for (i = 0; i <= 7; i = i + 1)

if (data_in[i] == 1)
count= count + 1;

end

endtask

endmodule

```

A task is declared by name 'count\_1s\_in\_byte' and the keyword task is used to define the task.

Task ends with keyword endtask.

A for loop is used to count the string 'data\_in' inputs. When the input data\_in[i] is '1' then the count is incremented by 1.

**Example 7.6** Verilog RTL for the task

```

module count_one_function (data_in, out);
input [7:0] data_in;
output reg [3:0] out;
always @(data_in)
out = count_1s_in_byte(data_in);
always @(data_in)
out = count_1s_in_byte(data_in);
// function declaration from here.
function [3:0] count_1s_in_byte(input [7:0] data_in);
integer i;
begin
count_1s_in_byte = 0;
for (i = 0; i <= 7; i = i + 1)
if (data_in[i] == 1) count_1s_in_byte = count_1s_in_byte + 1;
end
endfunction
endmodule

```



A function is declared by name 'count\_1s\_in\_byte' and the keyword function is used to define the function.

A function ends with keyword endfunction.

A for loop is used to count the string 'data\_in' inputs. When the input data\_in[i] is '1' then the count\_1s\_in\_byte is incremented by 1.

**Example 7.7** Verilog RTL with function calls

## 7.3 Parity Generators and Detectors

In most of the practical ASIC and SOC designs, Verilog RTL is used to describe the protocol behavior. The requirement and objective is functional correctness of the design and timing and cycle accurate models. In most of the practical applications, the parity needs to be detected to report for even parity or odd parity. If even number of 1's is there in any string then the parity is treated as even parity and if odd number of 1's are there in the string then parity will be treated as odd parity. This section focuses on the parity generator and checker.

### 7.3.1 Parity Generator

Efficient Verilog RTL is described in Example 7.8. As described in the RTL the even or odd parity is generated at output "q\_out". Even parity is indicated by logic '0' and odd parity is indicated by logic '1'.

```

module parity_gen (data_in, clk, q_out);

input data_in;

input clk;

output reg q_out;

reg even_odd; // The state t indicate even or odd parity

parameter EVEN=0, ODD=1;

always @ (posedge clk)
case (even_odd)

EVEN: begin

q_out <= data_in ? 1 : 0;

even_odd <= data_in ? ODD : EVEN;

end

ODD: begin

q_out <= data_in ? 0 : 1;

even_odd <= data_in ? EVEN : ODD;

end

endcase

endmodule

```

The procedural 'always' block is triggered on the positive edge of clock 'clk'.

The case construct is used to generate parallel logic depending on the 'even\_odd' condition.

For 'even\_odd=0' the EVEN state is executed and q\_out is assigned depending on the expression defined.

For 'even\_odd=1' the ODD state is executed and q\_out is assigned depending on the expression defined.

**Example 7.8** Verilog RTL for the parity generator

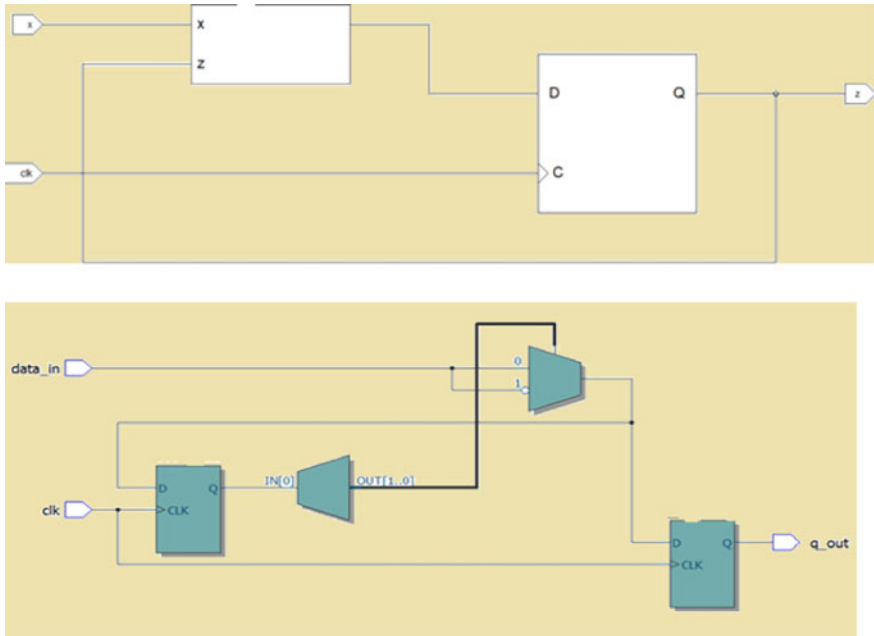


Fig. 7.9 Synthesized logic for parity generator

The synthesized result is shown in Fig. 7.9 and is register logic with the combinational logic at the data input of flip-flop. Multiple registers can be inferred by the synthesizer depending on the number of nonblocking assignments inside the edge sensitive procedural “always” block.

### 7.3.2 Add\_Sub\_Parity Checker

Consider the practical scenario in the design to use the multiple functional blocks. The design requirement is to complement input “add\_sub” when “add\_sub=1” then perform the complement of an input ‘b’. For “add\_sub=0” pass input ‘b’ as it is. The adder operates on two operands and the result from the complement logic. Adder generates an output “cy\_out and sum”. The parity checker is used at the output stage to find the even number of 1’s or odd number of 1’s in the string.

```

module add_sub_parity (p, a, b, add_sub);

input [7:0] a, b;

input add_sub; // 0 for add, 1 for subtract

output p; // parity of the result

wire [7:0] Bout, sum; wire carry;

complementor M1 (Bout, b, add_sub);

adder M2 (sum, carry, a, Bout, add_sub);

parity_checker M3 (p, {carry, sum});

endmodule

```

The module instantiation of 'complementor', 'adder' and 'parity checker' is described in the top level module 'add\_sub\_parity'

```

module parity_checker (out_par, in_word);

input [8:0] in_word;

output reg out_par;

always @ (in_word)

out_par = ^ (in_word);

endmodule

```

Combinational logic functionality for the 'parity\_checker' is coded by using bit wise XOR operator. For even number of 1's in the string it generates a result 'out\_par=0'.

For odd number of 1's in the string it generates a result 'out\_par=1'.

**Example 7.9** Verilog RTL for ADD-SUB parity

```

module adder (sum, cy_out, in1, in2, cy_in);

input [7:0] in1, in2;

input cy_in;

output [7:0] sum; reg [7:0] sum;

output cy_out; reg cy_out;

always @ (in1 or in2 or cy_in)

{cy_out, sum} = in1 + in2 + cy_in;

endmodule
    
```



Combinational multi-bit adder is coded by using level sensitive procedural 'always' block.

The output 'cy\_out' is carry output from the adder and 'sum' is 8-bit output from the adder.

```

module complementor (Y, X, comp);

input [7:0] X;

input comp;

output [7:0] Y;

reg [7:0] Y;

always @ (X or comp)

if (comp)

Y = ~X;

else

Y = X;

endmodule
    
```

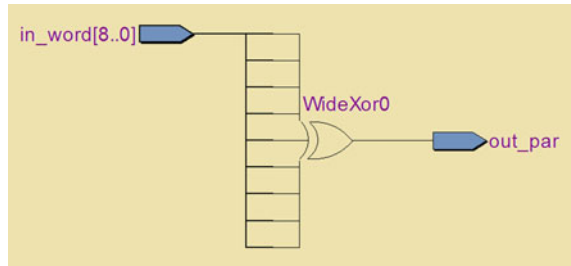


The complement output y is generated by the complement of 8-bit input x.

Procedural 'always' block is used which is level sensitive to the changes on the flag input 'comp' and input 'x'.

**Example 7.9** (continued)

**Fig. 7.10** Synthesized  
ADD-SUB parity checker



The Verilog RTL is described by using RTL as shown in Example 7.9. Inputs for the logic are a, b, add\_sub, and an output is ‘p’.

The partial synthesized logic is shown in Fig. 7.10. The overall synthesized logic consists of three blocks “complementor”, “adder”, and a “parity\_checker”.

## 7.4 Barrel Shifters

In most of the DSP applications the combinational shifters are used to perform the shifting operations on the data input. The combinational shifters are called as barrel shifter. The advantage of barrel shifter is that it performs the shifting operation depending on the required number of bits or control inputs without any clocked logic. Most of the barrel shifters are designed by using the multiplexer logic.

Example 7.10 is described in below and has 8-bit input “d\_in”, three-bit control input “c\_in” and an 8 bit output “q\_out”. The synthesis outcome is shown in the Fig. 7.11.



```
module barrel_shifter(d_in,q_out,c_in); // Main module of 8-Bit Barrel shifter

    input [7:0]d_in;

    output [7:0]q_out;

    input[2:0]c_in;

    mux m1(q_out[0],d_in,c_in);

    mux m2(q_out[1],{d_in[0],d_in[7:1]},c_in);

    mux m3(q_out[2],{d_in[1:0],d_in[7:2]},c_in);

    mux m4(q_out[3],{d_in[2:0],d_in[7:3]},c_in);

    mux m5(q_out[4],{d_in[3:0],d_in[7:4]},c_in);

    mux m6(q_out[5],{d_in[4:0],d_in[7:5]},c_in);

    mux m7(q_out[6],{d_in[5:0],d_in[7:6]},c_in);

    mux m8(q_out[7],{d_in[6:0],d_in[7:7]},c_in);

endmodule

module mux(y,d,c); // Sub module of 8-Bit barrel shifter

    input[7:0]d;
```

Eight multiplexer instances using primitive 'mux'.  
Instances are named from m1 to m8.



**Example 7.10** Verilog RTL for barrel shifter

```
output y;

reg y;

input [2:0]c;

always @ (c)

begin

  case (c)

    3'b000 : y = d[0];

    3'b001 : y = d[1];

    3'b010 : y = d[2];

    3'b011 : y = d[3];

    3'b100 : y = d[4];

    3'b101 : y = d[5];

    3'b110 : y = d[6];

    3'b111 : y = d[7];

    default : y = 1'b0;

  endcase

end

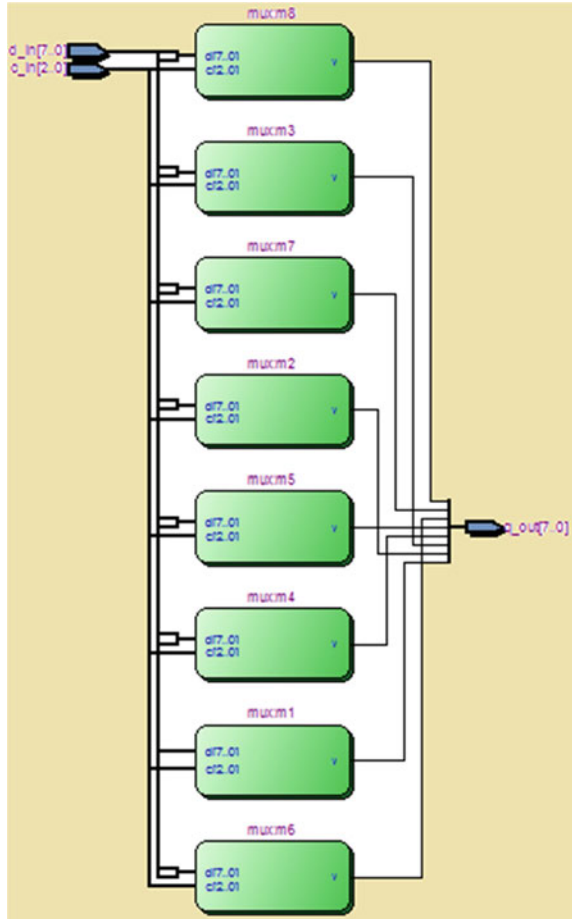
endmodule
```



The procedural 'always' block is sensitive to the level changes 'c' and used to infer the 8:1 MUX using the 'case' construct.

**Example 7.10** (continued)

**Fig. 7.11** Synthesized barrel shifter



## 7.5 Summary

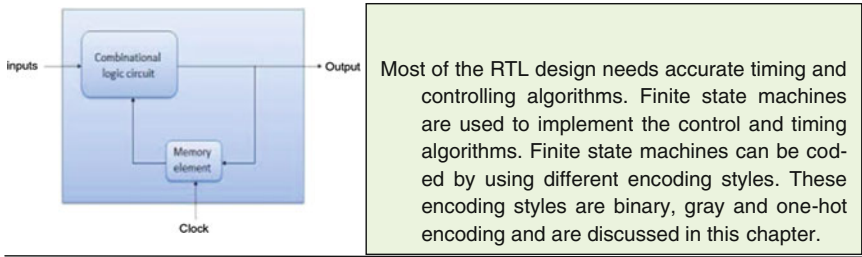
The following are key points to summarize this chapter:

1. The design partitioning can give the good and clear visibility of the data and control paths for the ASIC design.
2. The Verilog RTL for the complex design should have the separate modules for the data paths and control paths.
3. Use the resource sharing concepts while coding for the logic unit. All the logical operations can be performed by using full adder component with additional combinational logic.
4. Do not use the function and task while writing the synthesizable code.
5. Function does not consist of delays or timing control constructs. Task can consists of timing control and delay constructs.

6. Parity generators are used to generate an even or an odd parity for the data input string.
7. Barrel shifters are combinational shifters and designed by using mux-based logic.

# Chapter 8

## Finite State Machines



**Abstract** Finite state machine (FSM) is source synchronous sequential designs where every register is triggered on the active edge of clock. The two types of state machines are Moore and Mealy. This chapter discusses about the efficient and synthesizable FSM coding using Verilog RTL. The key differences between the Moore and Mealy machines as well as different FSM encoding styles are discussed in detail. This chapter illustrates the Verilog RTL examples with the multiple ‘always’ blocks to represent the efficient state machines. This chapter also focuses on the do’s and don’ts while coding FSM. The FSM design performance improvement with the key guidelines is also described in this chapter.

**Keywords** FSM • Moore • Mealy • Binary encoding • Gray encoding • One-hot encoding • State register • Current state • Next state • State transition table • State diagrams • State transition diagrams • Output combinational logic • Level to pulse conversion • Glitch free output • STA • Timing path • Sequential logic • Sequence detector

The finite state machine (FSM) is a very important design block in the ASIC design. Most of the ASIC designs and controller design needs the efficient and synthesizable state machines and commonly called as FSM. The FSMs can be described very efficiently by using the Verilog HDL and for ASIC design engineer, it is required to have in-depth understanding about the efficient coding of state machines.

Basically, FSMs are predefined sequences on the preordered or defined events and are source synchronous designs. FSMs can be coded efficiently for the synthesizable outcome using the multiple- or single-procedural block. In the practical scenario, it is recommended to use the multiple-procedural blocks to describe the state machines. One of the procedural blocks can describe the combinational logic and level sensitive to the inputs or the states. Whereas, the other procedural block can be edge sensitive to the positive edge of clock or to the negative edge of clock.

The multiple-procedural block FSM is better for the readability and can generate efficient synthesis results. The main objective of this chapter is to describe the efficient and synthesizable Verilog coding styles to describe the FSMs.

## 8.1 Moore Versus Mealy Machines

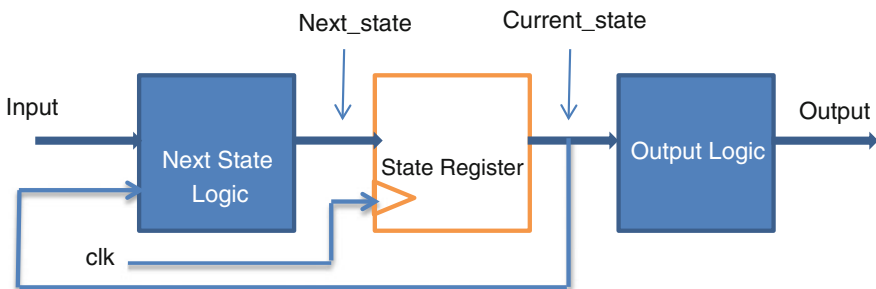
FSMs are classified as Moore and Mealy machines. In the Moore machines, the output is the function of the present or current state and in the Mealy machine, the output of FSM is the function of the present or current state as well as present input.

In the Moore machine, an output is stable for one clock cycle and hence output is glitch or hazard free. In the Mealy machines, an output may or may not be stable for one clock cycle as it is the function of current state or change in the input.

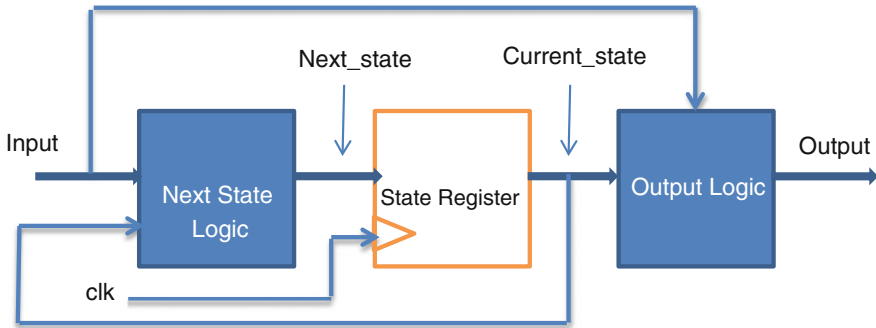
The timing analysis for the Moore machine is very simple due to clean register-to-register path but for the Mealy machine there might be chances of timing violations if the input changes during the setup time window.

But the disadvantage of Moore machine is it needs more number of states compared to Mealy machine. Practical scenario is that the Mealy machine has one state less compared to Moore machine.

Figure 8.1 describes the internal structure for the machine and it consists of combinational block as next state logic which is dependent upon the 'Current\_state' and an input, state register block which is dependent on the 'Next\_state' and output logic block which is purely combinational in nature and depends upon the 'Current\_state.' As discussed earlier, Moore machine output is the function of 'Current\_state' and hence stable for one clock cycle.



**Fig. 8.1** Block diagram of Moore machine



**Fig. 8.2** Block diagram of Mealy machine

Figure 8.2 describes the internal structure for the mealy machine and it consists of combinational block as next state logic which is dependent upon the ‘Current\_state’ and an input, state register block which is dependent on the ‘Next\_state’ and an output logic block which is purely combinational in nature and depends upon the ‘Current\_state’ as well changes in the input. As discussed earlier, Mealy machine output is the function of ‘Current\_state’ as well as changes in the inputs and hence may or may not be stable for one clock cycle. Due to this, the Mealy machines are prone to glitches.

How to code an efficient FSM is one of the important points to discuss. As an ASIC design engineer, the overall efficiency of the design is dependent upon the efficient Verilog RTL coding. Most of the inexperienced ASIC engineers uses single-procedural ‘always’ block for describing the behavior of the FSM. But single ‘always’ block FSM always leads to inefficient coding and creates issue while synthesizing the design and even during timing analysis.

In the practical scenarios two- or three-procedural block FSMs are used. In this chapter, I have recommended to use the three-procedural block FSM. Multiple-procedural block FSM increases the number of lines of code but is the most efficient during synthesis and timing analysis. Even this improves the overall readability and reusability during the reviews and design cycle.

1. One of the procedural ‘always’ blocks describes the functionality for the ‘Next State Logic’ and it is level sensitive to changes on the inputs and ‘Current\_state.’
2. Another procedural ‘always’ block is used to describe the register logic and sensitive to positive or negative edge of clock and hence used to infer the state register sequential logic.
3. Third procedural ‘always’ block is sensitive to the changes on ‘Current\_state’ and used to infer the combinational logic. This is true for the Moore machine
4. For the Mealy machine, third procedural ‘always’ block is sensitive to the changes on ‘Current\_state’ as well as input and used to infer the combinational logic.

**Table 8.1** Differences between Moore and Mealy machines

Moore machine	Mealy machine
Outputs are function of current state only	Outputs are function of the current state and inputs also
As output is the function of current state it is stable for one clock cycle	Output is the function of current state and inputs so it may change during the state and hence may or may not be stable for one clock cycle
Output is stable for one clock cycle and not prone to glitches or spikes	Output may change multiple times depending on changes in the input and hence prone to glitches or hazards
It requires more number of states compared to Mealy machine	Mealy machine needs at least one state less compared to Moore machine
STA is easy as combinational paths between the registers are shorter	STA is complex as combinational paths are relatively larger area compared to Moore machine
Higher operating frequency compared to Mealy machine	Less operating frequency compared to Moore machine

Table 8.1 illustrates the key differences between Moore and Mealy machines.

The template shown in Fig. 8.3 gives the information about the steps and declaration for the FSM coding.

The practical FSM for the toggle flip-flop is described using the three-procedural block FSM. The example describes the efficient Verilog RTL for Table 8.2. The state table is used to describe the state transition on clock edge (Example 8.1).

The synthesized logic for the toggle flip-flop using FSM is shown in Fig. 8.4 and it infers the state register triggered on the positive edge of clock and has active low asynchronous reset ‘reset\_n’. Due to use of the ‘case’ construct, the decoding logic is inferred, treat this logic as a ‘Not’ gate. The output is generated at ‘y\_out’ and the output toggles on every positive edge of clock ‘clk.’

### 8.1.1 Level to Pulse Converter

The level to pulse converter partial state transition diagram is shown in Fig. 8.5. As shown in the diagram the FSM remains in the state ‘S0’ for the input data\_in=0 and for the data input data\_in=1 it remains in the state ‘S1.’ The state transition table for the Mealy level to pulse converter is shown in Table 8.3.

The synthesizable Verilog RTL using three-procedural block is described in Example 8.2.

As described in the Verilog RTL, the output of level to pulse converter is the function of an input ‘data\_in’ and ‘current\_state.’ The Verilog RTL generates the structure shown in Fig. 8.6 for the level to pulse converter.

The synthesized logic for the Mealy level to pulse converter is shown in Fig. 8.7 and it infers the register logic with the combinational structure at output. Thus, the output of Mealy level to pulse converter is function of the ‘current\_state’ and an input ‘data\_in.’



1. Declare module with the FSM name and list the inputs and outputs.  
**module FSM\_NAME( // input output list);**
2. Declare the state variables using 'parameter', For example if FSM has two states then declare  
**Parameter s0;**  
**parameter s1;**
3. Declare the storage data types 'reg' for the next state and current state. For Example for one bit data type declare:  
**reg current\_state;**  
**reg next\_state;**
4. Describe the state register logic sensitive to the edge, for example:  
**always@ (posedge clk or negedge reset\_n)**  
**begin**  
  
**//Functionality**  
  
**end**  
  
**//Use non-blocking assignment to code the sequential logic**
5. Describe the next state logic which is sensitive to level, for example  
**always @ (current\_state, input)**  
**begin**  
**case (current\_state)**  
**//Functionality of the next state logic**  
**endcase**  
**end**  
**//Use the blocking assignments to code the combinational logic**
6. Describe the output logic which is sensitive to level, for example  
**always @ (current\_state)// For Moore machine**  
**always@(current\_state, input) // For Mealy machine**  
**begin**  
**case (current\_state)**  
**//Functionality of the next state logic**  
**endcase**  
**end**  
**//Use the blocking assignment to code the combinational logic**

**Fig. 8.3** Steps for efficient FSM Verilog RTL coding

**Table 8.2** State transition table for the toggle flip-flop

Current_state	Next_state
0	1
1	0

```
//FSM for the toggle flip-flop
module toggle_flip_flop_fsm ( clk, reset_n, y_out );
input clk;
input reset_n;
output reg y_out;
parameter s0=0;
parameter s1=1;
reg current_state;
reg next_state;

//State register logic
always@ (posedge clk or negedge reset_n)
begin
if (~reset_n)
current_state <= s0;
else
current_state <= next_state;
end

//Next state combinational logic

always @ (current_state)
begin
case (current_state)
s0 : next_state = s1;
s1 : next_state =s0;
default : next_state =s0;
endcase
end

//Output combinational logic
always@ (current_state)
case ( current_state)
s0 : y_out = 1'b0;
s1 : y_out =1;
default : y_out=1'b0;
endcase
endmodule
```

The input and outputs are declared for the toggle flip-flop.

Inputs are named as 'clk', 'reset\_n' and an output is named as 'y\_out'.

State parameters are declared as 's0', 's1'. By using the data type 'reg' the 'current\_state' and 'next\_state' are declared.

The state register sequential block is sensitive to positive edge of clock.

During reset condition 'current\_state' is assigned to 's0' and during normal operation the 'next\_state' is assigned to 'current\_state'

The next state logic combinational block is sensitive to the 'current\_state' for the toggle flip-flop.

The functionality is described by using the 'case' construct'. Please refer the state transition table

The output combinational logic is function of the 'current\_state' and the functionality is described using the 'case' construct. During state s0 the output 'y\_out' is assigned to logic '0' and during state s1 the output 'y\_out' is assigned to logic '1'. So it infers the toggle flip-flop.

**Example 8.1** Verilog RTL for toggle flip-flop

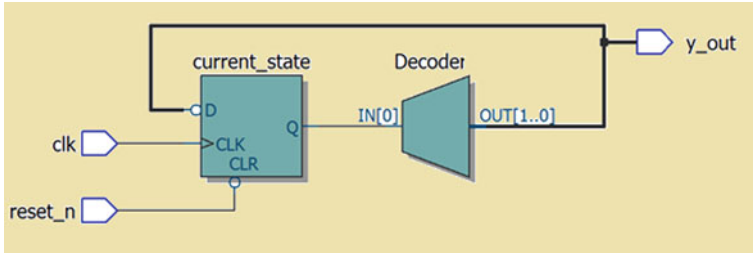


Fig. 8.4 Synthesized toggle flip-flop

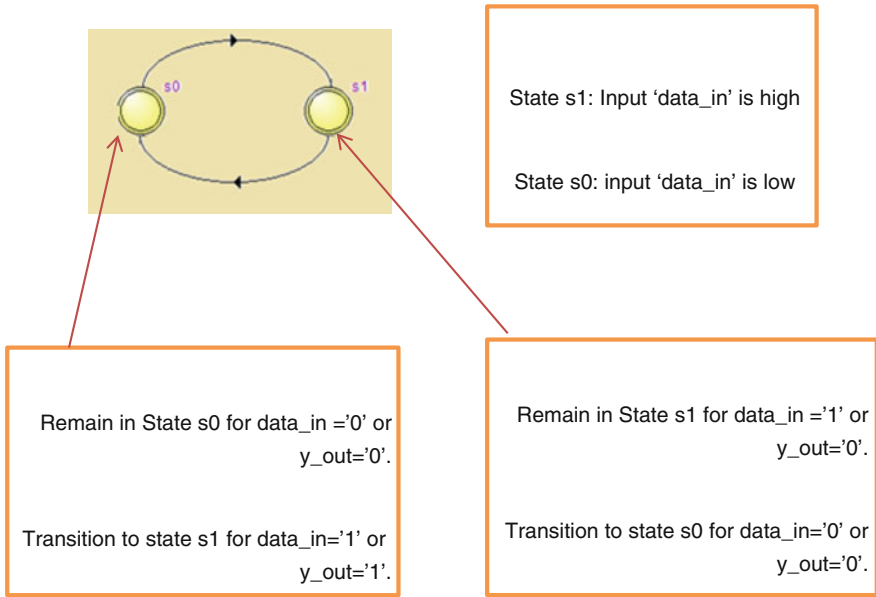


Fig. 8.5 Partial state transition diagram for Mealy level to pulse converter

Table 8.3 State transition table for the Mealy level to pulse converter

Data_in	Current_state	Next_state	y_out
0	s0	s0	0
1	s0	s1	1
0	s1	s0	0
1	s1	s1	0

```

//Synthesizable RTL for Mealy level to pulse converter
module level_pulse ( data_in, clk, reset_n, y_out );
input clk;
input reset_n;
input data_in;
output reg y_out;

parameter s0=0;
parameter s1=1;

reg current_state;
reg next_state;

//State register logic
always@ (posedge clk or negedge reset_n)
begin

if (~reset_n)
    current_state <= s0;
else
    current_state <= next_state;
end

//Next state logic combinational block
always @ (current_state, data_in)
begin
case (current_state)
s0 : if (data_in) next_state = s1;
     else next_state=s0 ;
s1 : if (data_in) next_state = s1;
     else next_state=s0 ;
default : next_state =s0;

endcase
end

//Output logic combinational block
always@ (current_state, data_in)
case (current_state)
s0 : y_out = 1'b0;
s1 : if ( data_in) y_out =1;
     else y_out = 1'b0;
default : y_out=1'b0;
endcase
endmodule
    
```

The input and outputs are declared for level to pulse converter.

Inputs are named as 'clk', 'reset\_n' and 'data\_in'. The output is named as 'y\_out'.

State parameters are declared as 's0', 's1'. By using the data type 'reg' the 'current\_state' and 'next\_state' are declared.

The state register sequential block is sensitive to positive edge of clock.

During reset condition 'current\_state' is assigned to 's0' and during normal operation the 'next\_state' is assigned to 'current\_state'

The next state logic combinational block is sensitive to the 'current\_state' for the level to pulse converter.

The functionality is described by using the 'case' construct'. Please refer the state transition table

The output combinational logic is function of the 'current\_state' as well as on 'data\_in' and the functionality is described using the 'case' construct. For 'data\_in=1' and state s1 as current\_state an output 'y\_out' is assigned to logic '1'. In the default state an output 'y\_out' is logic '0' and the default state is reset state or initialization state for the FSM.

**Example 8.2** Verilog RTL for level to pulse converter

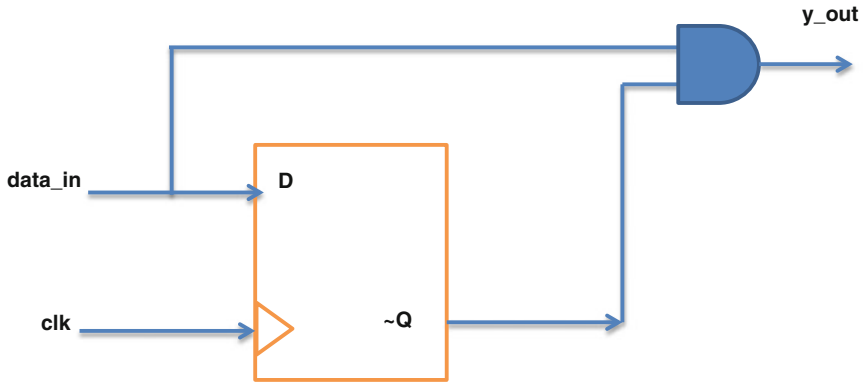


Fig. 8.6 Level to pulse convert logic diagram

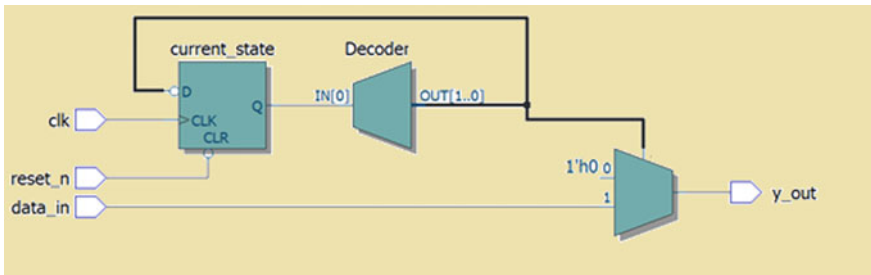


Fig. 8.7 Synthesized logic for level to pulse converter

## 8.2 FSM Encoding Styles

FSM can be described by many styles and practically there are three encoding styles used to describe the FSMs. These styles are named as

- a. *Binary Encoding* FSM can be described by using binary encoding styles and by using this style, the number of register elements used is equal to  $\log_2$  number of states. Consider an FSM has four states; then, the number of registers equal to  $\log_2 4$  is equal to 2.
- b. *Gray Encoding* FSM can be efficiently described by using Gray encoding technique and in this style the gray codes are used to represent the states. The number of register elements used is equal to  $\log_2$  number of states. Consider an FSM has four states; then, the number of registers equal to  $\log_2 4$  is equal to 2.
- c. *One-hot encoding* FSMs can be efficiently described using one-hot encoding style. One-hot indicates that only one bit is active high at a time or hot at a time. The number of register elements used is equal to number of states in the FSM. Consider an FSM has four states then the number of registers also equals 4. This

**Table 8.4** FSM encoding style comparison

FSM encoding	Binary	Gray	One-hot
Representation	Binary number	Gray number	16 bit one-hot number
No. of registers	4	4	16
Area	Same as gray encoding	Same as binary encoding	More

**Table 8.5** FSM State representation

FSM states	Binary	Gray	One-hot
S0	00	00	0001
S1	01	01	0010
S2	10	11	0100
S3	11	10	1000

style requires more area but advantage is it has clean register-to-register path and it makes STA very simple. If FSM has 16 states then one-hot encoding needs 16 flip-flops.

The comparison of different FSM styles for 16 states is shown in Table 8.4. The encoding representation for 4-state FSM is shown in Table 8.5.

### 8.2.1 Binary Encoding

As discussed earlier, the binary encoding style can be used if the area requirement is a constraint on the design. In this encoding style state parameters for the binary encoding are represented in the binary format.

#### 8.2.1.1 Two-Bit Binary Counter FSM

Two-bit binary counter FSM is described in Example 8.3. As described in the example, the number of states is equal to 4 and it needs four state variables ‘s0,’ ‘s1,’ ‘s2,’ and ‘s3.’ The number of flip-flops used to represent the functionality of counter is equal to 2.

The state transition table is shown in Table 8.6. The state transition diagram is shown in Fig. 8.8. The transition from one state to another state occurs on the positive edge of clock. Default state is ‘s0’ and it is the reset state.

The synthesized logic for the two-bit binary counter is shown in Fig. 8.9. As shown in figure, the state register is triggered on the positive edge of the clock and has active low asynchronous reset ‘reset\_n.’ The output combinational logic is a decoding structure due to the ‘case’ construct.’ The synthesis outcome is Moore machine as the output is function of the Current\_state only.

```

module binary_2_bit_counter ( clk, reset_n, y_out);
input clk;
input reset_n;
output reg [1:0] y_out;
parameter s0=2'b00;
parameter s1=2'b01;
parameter s2 =2'b10;
parameter s3 = 2'b11;
reg [1:0] current_state;
reg [1:0] next_state;

//State register logic
always@ (posedge clk or negedge reset_n)
begin
if (~reset_n)
    current_state <= s0;
else
    current_state <= next_state;
end

//Next state combinational logic
always @ (current_state)
begin
case (current_state)
s0 : next_state = s1;
s1 : next_state =s2;
s2 : next_state =s3;
s3 : next_state =s0;
default : next_state =s0;
endcase
end

//Output combinational logic
always@ (current_state)
case ( current_state)
s0 : y_out = 2'b00;
s1 : y_out = 2'b01;
s2 : y_out = 2'b10;
s3 : y_out = 2'b11;
default : y_out=2'b00;
endcase
endmodule
    
```

The state parameters are declared as 's0', 's1', 's2', 's3'.

The binary encoding style is used and the reg data type is used to describe the 'current\_state' and 'next\_state'.

The state register sequential block is sensitive to positive edge of clock.

During reset condition 'current\_state' is assigned to 's0' and during normal operation the 'next\_state' is assigned to 'current\_state'

The next state logic combinational block is sensitive to the 'current\_state' for the two bit binary counter

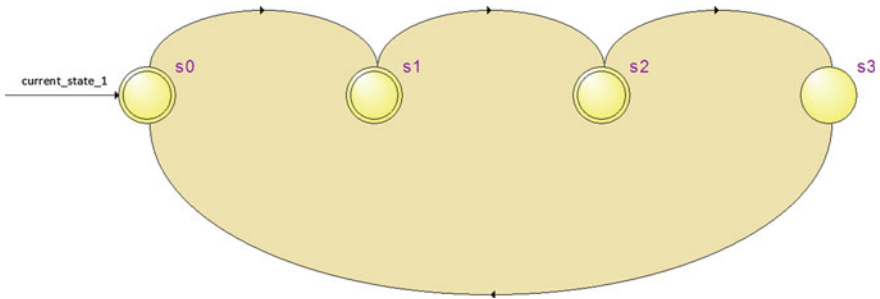
The functionality is described by using the 'case' construct'. Please refer the state transition table for the transition. Default state is 's0'.

The output combinational logic is function of the 'current\_state' and the functionality is described using the 'case' construct. For example state s1 as current\_state an output 'y\_out' is assigned to binary '01'. In the default state an output 'y\_out' is binary '00' and the default state is reset state or initialization state for the FSM.

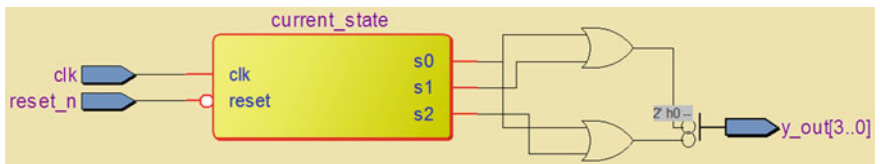
**Example 8.3** Verilog RTL for two-bit binary counter

**Table 8.6** State transition table for binary encoding

Current_state	Next_state
s0=00	s1=01
s1=01	s2=10
s2=10	s3=11
s3=11	s0=00



**Fig. 8.8** State transition diagram for two-bit binary counter



**Fig. 8.9** Synthesized logic for two-bit counter

## 8.2.2 Gray Encoding

As discussed earlier, the Gray encoding style can be used if the area requirement is a constraint on the design. In this encoding style, state parameters are represented in the Gray format.

### 8.2.2.1 Two-Bit Gray Counter FSM

Two-bit Gray counter FSM is described in Example 8.4. As described in example, the number of states is equal to 4 and it needs four state variables ‘s0,’ ‘s1,’ ‘s2,’ and ‘s3.’ The number of flip-flops used to represent the functionality of counter is equal to 2.

The state transition table is shown in Table 8.7. The state transition diagram is shown in Fig. 8.10. The transition from one state to another state occurs on the positive edge of clock. Default state is ‘s0’ and it is the reset state.



```
//FSM for 2-bit gray counter
module gray_counter ( clk, reset_n, y_out);
input clk;
input reset_n;
output reg [1:0] y_out;
parameter s0=2'b00;
parameter s1=2'b01;
parameter s2 =2'b10;
parameter s3 = 2'b11;
reg [1:0] current_state;
reg [1:0] next_state;
//State register logic
always@ (posedge clk or negedge reset_n)
```

The state parameters are declared as 's0', 's1', 's2', 's3'.

The gray encoding style is used and the reg data type is used to describe the 'current\_state' and 'next\_state'.

The state register sequential block is sensitive to positive edge of clock.

During reset condition 'current\_state' is assigned to 's0' and during normal operation the 'next\_state' is assigned to 'current\_state'

```
begin
if (~reset_n)
current_state <= s0;
else
current_state <= next_state;
end
//Next state combinational logic
always @ (current_state)
begin
case (current_state)
s0 : next_state = s1;
s1 : next_state =s3;
s3 : next_state =s2;
s2 : next_state =s0;
default : next_state =s0;
endcase
end
//Output combinational logic
always@ (current_state)
case ( current_state)
s0 : y_out = 2'b00;
s1 : y_out = 2'b01;
s3 : y_out = 2'b11;
s2 : y_out = 2'b10;
default : y_out=2'b00;
endcase
endmodule
```

The next state logic combinational block is sensitive to the 'current\_state' for the two bit binary counter

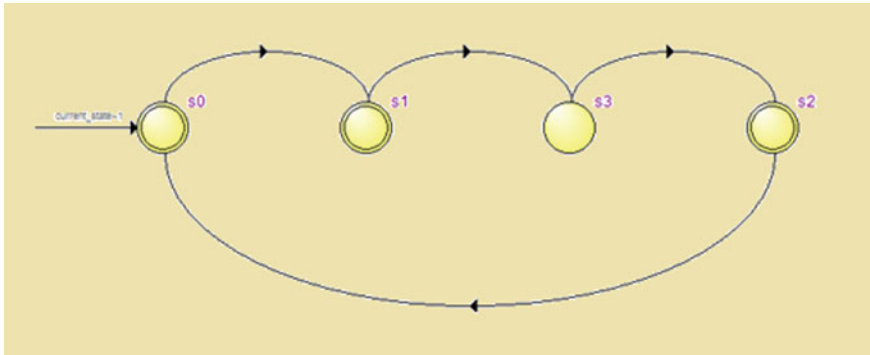
The functionality is described by using the 'case' construct'. Please refer the state transition table for the transition. Default state is 's0'.

The output combinational logic is function of the 'current\_state' and the functionality is described using the 'case' construct. For example state s3 as current\_state an output 'y\_out' is assigned to binary '11'. In the default state an output 'y\_out' is binary '00' and the default state is reset state or initialization state for the FSM.

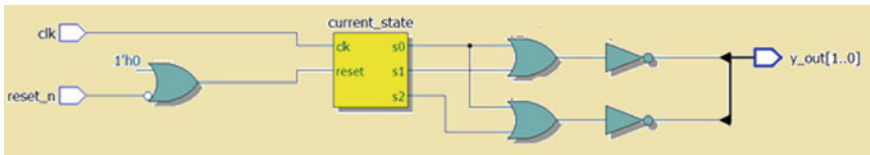
Example 8.4 Verilog RTL with gray encoding style

**Table 8.7** State transition table for gray encoding

Current_state	Next_state
s0=00	s1=01
s1=01	s3=11
s3=11	s2=10
s2=10	s0=00



**Fig. 8.10** State transition diagram for gray encoding



**Fig. 8.11** Synthesized logic for the gray encoding style

The synthesized logic for the two-bit Gray counter is shown in Fig. 8.11. As shown in figure, the state register is triggered on the positive edge of the clock and has active low asynchronous reset 'reset\_n.' The output combinational logic is decoding structure due to the 'case' construct.'

### 8.3 One-Hot Encoding

Two-bit counter FSM using one-hot encoding is described in Example 8.5. As described in example, the number of states is equal to 4 and it needs four state variables 's0,' 's1,' 's2,' and 's3.' The number of flip-flops used to represent the functionality of counter is equal to 4.

The state transition table is shown in Table 8.8. The transition from one state to another state occurs on the positive edge of clock. Default state is 's0' and it is reset state.

```
//FSM for one-hot encoding
module one_hot_encoding ( clk, reset_n, y_out);
input clk;
input reset_n;
output reg [1:0] y_out;
parameter [3:0] s0=4'b0001;
parameter [3:0] s1=4'b0010;
parameter [3:0] s2 =4'b0100;
parameter [3:0] s3 = 4'b1000;
reg [3:0] current_state;
reg [3:0] next_state;
```

The state parameters are declared as 's0', 's1', 's2', 's3'.

The one-hot encoding style is used and the reg data type is used to describe the 'current\_state' and 'next\_state'. The reg is 4 bit wide.

```
//State register logic
always@ (posedge clk or negedge reset_n)
begin
if (~reset_n)
current_state <= s0;
else
```

The state register sequential block is sensitive to positive edge of clock.

During reset condition 'current\_state' is assigned to 's0' and during normal operation the 'next\_state' is assigned to 'current\_state'

```
current_state <= next_state;
end
```

```
//next state logic
always @ (current_state)
begin
case (current_state)
s0 : next_state =s1;
s1 : next_state =s2;
s2 : next_state =s3;
s3 : next_state =s0;
default : next_state =s0;
endcase
end
```

The next state logic combinational block is sensitive to the 'current\_state' for the two bit binary counter

The functionality is described by using the 'case' construct'. Please refer the state transition table for the transition. Default state is 's0'.

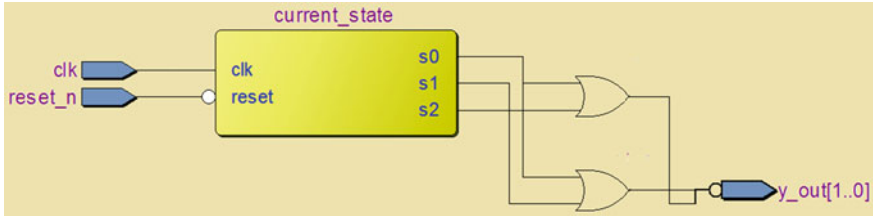
```
//Output combinational logic
always@ (current_state)
case ( current_state)
s0 : y_out = 2'b00;
s1 : y_out = 2'b01;
s2 : y_out = 2'b10;
s3 : y_out = 2'b11;
default : y_out=2'b00;
endcase
endmodule
```

The output combinational logic is function of the 'current\_state' and the functionality is described using the 'case' construct. For example state s3 as current\_state an output 'y\_out' is assigned to binary '11'. In the default state an output 'y\_out' is binary '00' and the default state is reset state or initialization state for the FSM.

Example 8.5 Verilog RTL for one-hot encoding

**Table 8.8** State transition table for one-hot encoding

Current_state	Next_state
s0=0001	s1=0010
s1=0010	s2=0100
s2=0100	S3=1000
s3=1000	s0=0001



**Fig. 8.12** Synthesized logic for one-hot encoding

The synthesized logic for the two-bit binary counter using one-hot encoding is shown in Fig. 8.12. As shown in figure, the state register is triggered on the positive edge of the clock and has active low asynchronous reset ‘reset\_n.’ This encoding method uses the four registers to represent the functionality. The output combinational logic is decoding structure to generate two-bit output using the ‘case’ construct.’

### 8.4 Sequence Detectors Using FSMs

FSMs are used to describe the functionality of sequence detectors. The efficient RTL coding using Verilog is used in the practical scenario to find out the correct sequence for the state. Depending on the requirements, either Moore or Mealy machines can be used to detect the correct sequence.

#### 8.4.1 Sequence Detector Using Mealy Machine Two Always Blocks

The state transition table for the sequence detector is shown with the required output (Table 8.9).

**Table 8.9** State transition table for sequence detector

Current_state	Input	Next_state	Output
S0=00	1	S1=01	2'b00
S1=01	0	S2=10	2'b01
S2=10	1	S3=11	2'b10
S3=11	0	S0=00	2'b11

```

module sequence_detector ( input clk, data_in, reset_n, output reg
    [1:0] y_out );
reg [1:0] state; // Declare state register
parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3; // Declare states

always @ (posedge clk or negedge reset_n) begin
if (~reset_n)
state <= S0;
else
case (state)
S0: if (data_in) begin
state <= S1;
end
else begin
state <= S0;
end
S1: if (~data_in) begin
state <= S2;
end
else begin
state <= S1;
end
S2: if (data_in) begin
state <= S3;
end
else begin
state <= S2;
end
S3: if (~data_in) begin
state <= S0;
end
else begin
state <= S3;
end
endcase
end

always @ (state or data_in)
begin
case (state)
S0: if (data_in) begin
y_out = 2'b00;
end
else begin
y_out = 2'b10;
end
S1: if (~data_in) begin
y_out = 2'b01;
end
else begin

```

The state register sequential block is sensitive to positive edge of clock.

During reset condition 'state' is assigned to 'S0' and during normal operation the 'state' is assigned to required state. Please refer the state transition table

The output combinational logic is function of the 'state' as well as data input 'data\_in' and the functionality is described using the 'case' construct. For example state s3 as current\_state an output 'y\_out' is assigned to binary '11' if data input 'data\_in=0', if 'data\_in=1' then output is assigned to '00'. In the default state an output 'y\_out' is binary '00' and the default state is reset state or initialization state for the FSM.

**Example 8.6** Verilog RTL for the sequence detector

```

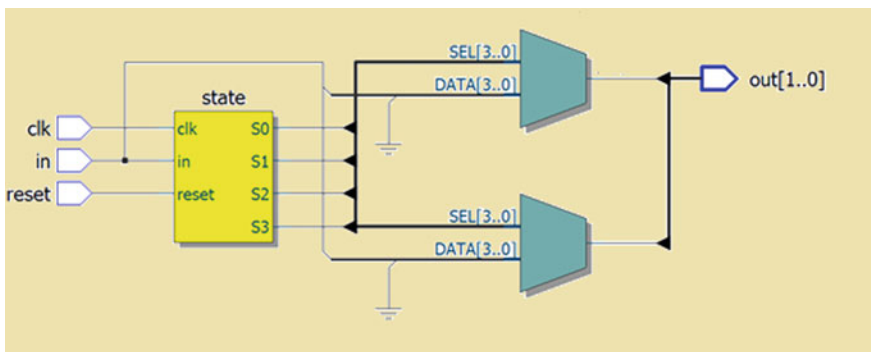
y_out = 2'b00;
end
S2: if (data_in) begin
y_out = 2'b10;
end
else begin
y_out = 2'b01;
end
S3: if (~data_in) begin
y_out = 2'b11;
end
else begin
y_out = 2'b00;
end
endcase
end
endmodule

```

**Example 8.6** (continued)

The synthesizable Verilog RTL is shown in Example 8.6 to detect the sequence 1010. The output of sequence detector is 2 bit and when sequence of 1010 is found then it generates output as “11”. To get the single output the output “11” can be given to AND logic.

The synthesizable Mealy machine sequence detector infers the state register sequential logic consisting of two registers and combinational decoding logic. In this, output is function of the state and input changes. The synthesized logic is shown in Fig. 8.13.



**Fig. 8.13** Synthesized logic for sequence detector

### 8.4.2 Sequence Detector Using Mealy Machine for '101' Sequence

Another Mealy machine sequence detector for the sequence 101 is shown in the state diagram Fig. 8.14. For the overlapping sequence of 10101 also, this works and generates an output 'y\_out = 1' after detecting the sequence '101.' The state transition table is shown.

As shown in Table 8.10 the Mealy machine output 'y\_out' is active high when input sequence '101' is detected. Output is function of the current state and changes in the input.

The Verilog RTL for the Mealy sequence detector is described in Example 8.7.

## 8.5 Improving the Design Performance for FSMs

The objective or goal for FSM coding is efficient synthesis and fast debugging. The reusability and modifications in the state encoding is another important point ASIC designer need to focus. Even the coding style should be compact as well as readable.

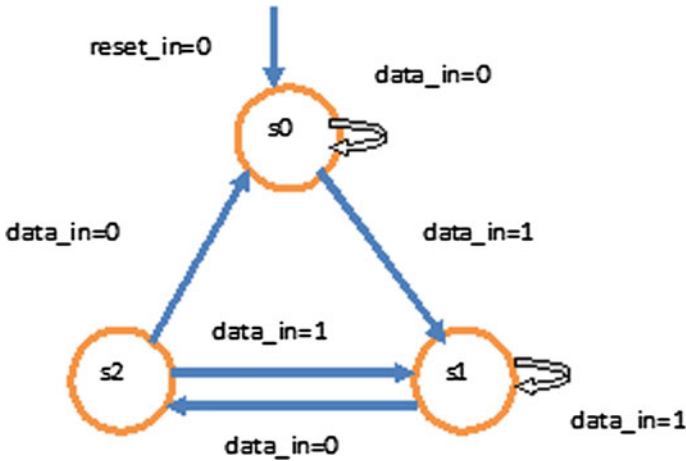


Fig. 8.14 Mealy machine state diagram for 101 sequence

Table 8.10 State transition table for sequence detector '101'

Input	Current _state	Next_state	Output
1	S0	S1	0
0	S1	S2	0
1	S2	S1	1

```

module sequence_detector ( clk, reset_n, data_in, y_out);
input clk;
input reset_n;
input data_in;
output y_out;
parameter s0=2'b00;
parameter s1=2'b01;
parameter s2 =2'b10;
reg [1:0] current_state;
reg [1:0] next_state;

```

The state register sequential block is sensitive to positive edge of clock.

During reset condition 'current\_state' is assigned to 's0' and during normal operation the 'next\_state' is assigned to 'current\_state'

```

//State register logic
always@ (posedge clk or negedge reset_n)
begin
if (~reset_n)
current_state <= s0;
else
current_state <= next_state;
end

```

```

//Next state combinational logic
always @ (current_state, data_in)
begin
case (current_state)
s0 : if (data_in) next_state = s1;
      else next_state=s0;
s1 : if (~data_in) next_state =s2;
      else next_state=s1;
s2 : if (data_in) next_state =s1;
      else next_state=s0;
default : next_state =s0;
endcase
end

```

The next state logic combinational block is sensitive to the 'current\_state' as well as 'data\_in' for the sequence detector.

The functionality is described by using the 'case' construct'. Please refer the state transition table for the transition. Default state is 's0'.

```

//Output combinational logic
always@ (current_state, data_in)
case ( current_state)
s0 : y_out = 0;
s1 : y_out = 0;
s2 : if (data_in) y_out=1;
      else y_out=0;
default : y_out=2'b0;
endcase
endmodule

```

The output combinational logic is function of the 'current\_state' and the functionality is described using the 'case' construct. For example state s2 as current\_state then if 'data\_in' is logic 1 then output 'y\_out' is asserted to logic '1'. The default state is reset state or initialization state for the FSM.

### Example 8.7 Sequence detector example to detect '101' sequence



The following are key guidelines used to improve the FSM performance.

- a. Do not use the single ‘always’ block FSM. As the issue is in readability and it does not yield in the efficient synthesis results.
- b. Use multiple-procedural block FSMs. In practical ASIC designs, two or three ‘always’ block FSMs are used as it improves the readability, reusability and it yield in the efficient synthesis results.
- c. Declare the state parameters according to the required state encoding and then declare the `next_state` and `current_state`.
- d. Use nonblocking assignments for describing the state register logic.
- e. Use blocking assignments for describing the next state combinational logic.
- f. Use blocking assignments for describing the output combinational logic.
- g. Use the ‘default’ condition in the ‘case’ construct to avoid the inference of latches.
- h. For use of ‘if-else’ construct the number of transitions in the state diagram should be same like number of ‘if-else’ clauses.
- i. Register the FSM outputs as it ensures that an output is glitch free.
- j. For better and efficient synthesis outcome use the one-hot encoding method.

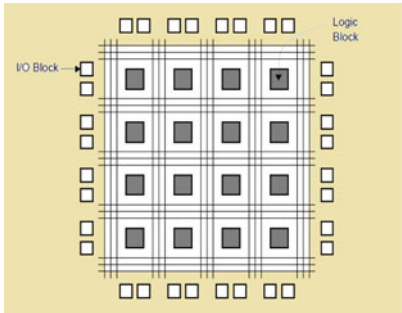
## 8.6 Summary

As discussed in this chapter, following are the key highlights to summarize

- a. FSMs are described very efficiently using Verilog RTL for better synthesizable outcome.
- b. FSMs are of two types: Moore and Mealy.
- c. In the Moore type FSM the output is the function of current state only.
- d. In the Mealy FSMs the output is the function of current state as well as inputs.
- e. FSM encoding styles are: Binary, Gray, and one-hot.
- f. One-hot encoding style is used for glitch free outputs and yields better and clean synthesis.
- g. In ASIC designs two or three always block FSMs are used to generate efficient synthesis.

# Chapter 9

## Simulation Concepts and PLD-Based Designs



Programmable Logic devices are used to realize the complex logic. Due to programmability the modern high density FPGAs are used to prototype the complex SOCs. This chapter discusses about the FPGA architecture, design flow and the simulation using the FPGA to realize the complex gate count designs. Even this chapter discusses about the design guidelines for FPGA based designs.

**Abstract** Programmable logic devices (PLDs) are used extensively in the research areas and even in the industrial applications to realize the complex designs due to programmability features. PLDs are used to prototype the ASIC SOCs due to the availability of the configurable logic blocks, multipliers, and DSP blocks. This chapter discusses about the PLD evolution, architecture of FPGA, and why to use FPGA, FPGA design guidelines and the logic realization using FPGAs. Even this chapter discusses about the simulation constructs and the different delays with the basic testbench.

**Keywords** PLD · CPLD · PAL · PLA · PROM · SPLD · FPGA · Programmable ASIC · LUT · CLB · IOB · Interconnect · Logic capacity · Logic density · DSP · Multiplier · Processor core · IO standards · Structured ASIC · Flash · SRAM · Antifuse · STA · RTL · Simulation · Intra-delay · Inter-delay · Combinational loop · Grouping · Clock gating · DLL · PLL · IOB · CLB · LUT · Interconnect · Clock skew · IP · XILINX · Spartan

### 9.1 Key Simulation Concepts

The design entered in the Verilog or VHDL needs to be simulated to check for the functional correctness of the design. For the HDL RTL functionality, the testbench need to be written using the nonsynthesizable Verilog constructs. Nonsynthesizable

Verilog constructs are used while developing the testbench. Please refer Appendix I for the synthesizable and nonsynthesizable Verilog constructs.

### 9.1.1 *Simulation for Blocking and Nonblocking Assignments*

Consider that the Verilog RTL consists of the blocking assignments shown in Example 9.1.

In the example, the procedural “always” block executes every time on the event on the clock “clk.” The “initial” block executes only once and is used to assign the value to ‘a,’ ‘b,’ ‘c,’ and ‘d.’ The simulation result for the nonblocking assignment is shown in Waveform 9.1.

Consider that the Verilog RTL consists of the nonblocking assignments shown in Example 9.2.

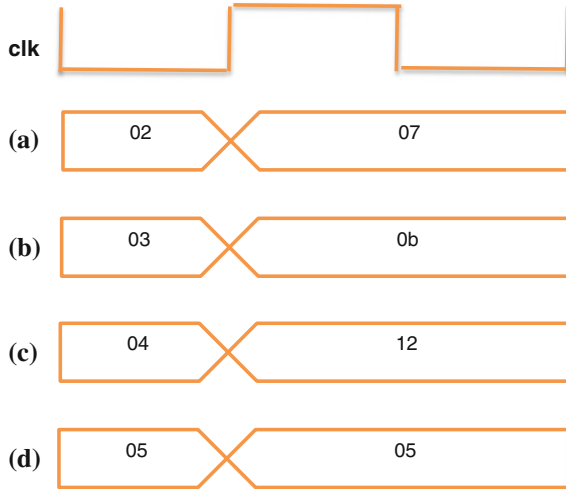
The simulation result for the above Verilog code using nonblocking is shown in Waveform 9.2.

```
reg [7:0] a,b,c,d;

always@(posedge clk)
begin
    a=b+c;
    b=a+d;
    c=a+b;
end

initial
begin
    a=8'h2;
    b=8'h3;
    c=8'h4;
    d=8'h5;
end
```

**Example 9.1** Simulation of Verilog blocking assignment

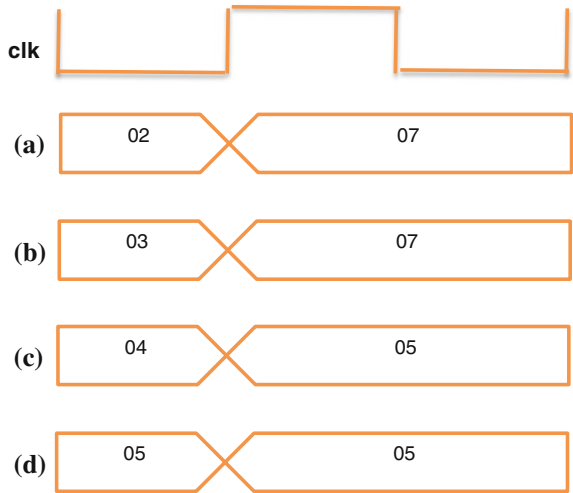


**Waveform 9.1** Simulation result for Verilog blocking assignment

```
reg [7:0] a,b,c,d;
always@(posedge clk)
begin
    a<=b+c;
    b<=a+d;
    c<=a+b;
end
initial
begin
    a=8'h2;
    b=8'h3;
    c=8'h4;
    d=8'h5;
end
```

**Example 9.2** Simulation for Verilog nonblocking assignment

**Waveform 9.2** Simulation result for Verilog nonblocking assignment



**Table 9.1** Difference between initial and always block

Initial	Always
In this, block assignment executes in the 0 simulation time and continues for the next specified sequence	In this, block assignments continues to execute in simulation time 0 and repeats forever depending on the sensitivity list event
This block is executed only once and the simulation stops at the end of this block	The simulation in this block continues forever. If wait construct is there then it will be held during simulation session
It is nonsynthesizable construct	It is synthesizable construct

The difference between the “initial” and “always” block is described in Table 9.1.

### 9.1.2 Blocking Assignments with Inter-assignment Delays

The inter-assignment delays with the blocking assignment, delay both the evaluation of the assignment and the update for the assignment.

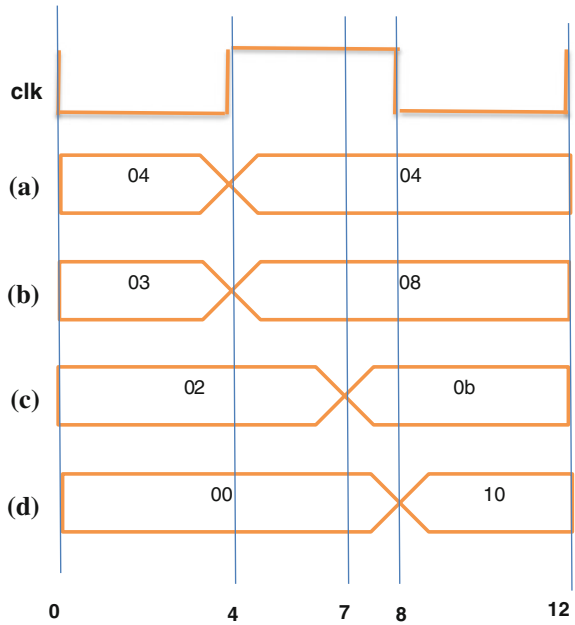
Consider the following Verilog code shown in Example 9.3.

Waveform 9.3 gives the simulation results for the blocking assignment with the inter-assignment delays.

```
always@(posedge clk)
begin
    b=a+a;
    #3 c=b+a;
    #1 d=c+a;
end
initial
begin
    a=4;
    b=3;
    c=2;
end
end
```

**Example 9.3** Verilog blocking assignment with inter-assignment delay

**Waveform 9.3** Simulation result for the Verilog blocking assignment with inter-assignment delay



### 9.1.3 *Blocking Assignments with Intra-assignment Delays*

The intra-assignment delays with the blocking assignment, delay the evaluation of the assignment but not the update for the assignment.

Consider the following Verilog code shown in Example 9.4.

Waveform 9.4 gives the simulation results for the blocking assignment with the intra-assignment delays.

### 9.1.4 *Nonblocking Assignments with Inter-assignment Delays*

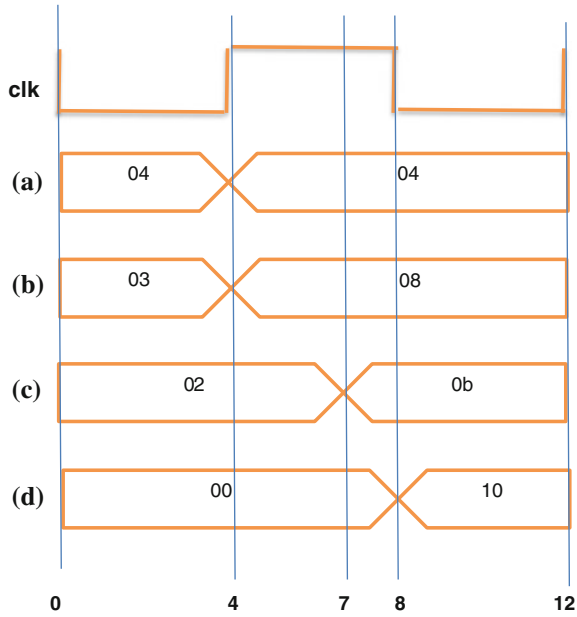
The intra-assignment delays with the nonblocking assignment delay both the evaluation of the assignment and the update for the assignment.

Consider the following Verilog code shown in Example 9.5.

Waveform 9.5 gives the simulation results for the nonblocking assignment with the inter-assignment delays.

```
always@(posedge clk)
begin
    b=a+a;
    c=#3 b+a;
    d=#1c+a;
end
initial
begin
    a=4;
    b=3;
    c=2;
end
```

**Example 9.4** Verilog blocking assignment with intra-assignment delay



**Waveform 9.4** Simulation result for the Verilog blocking assignment with intra-assignment delay

```
always@(posedge clk)
begin
    b<=a+a;
    #3 c<=b+a;

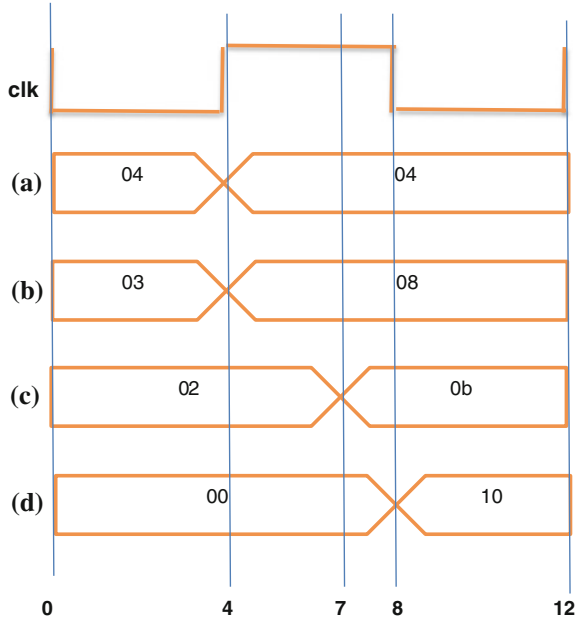
    #1 d<=c+a;
end

initial
begin
    a=4;
    b=3;
    c=2;
end
```

**Example 9.5** Verilog nonblocking assignment with inter-assignment delay



**Waveform 9.5** Simulation result for the Verilog nonblocking assignment with inter-assignment delay



**9.1.5 Nonblocking Assignments with Intra-assignment Delays**

The intra-assignment delays with the blocking assignment, the update of the assignment but not the evaluation of the assignment.

Consider the following Verilog code shown in Example 9.6.

Waveform 9.6 gives the simulation results for the blocking assignment with the intra-assignment delays.

**9.2 Simulation Using Verilog**

In Chaps. 1–8, we have discussed about the detail design synthesis and hardware inference. Verilog HDL is powerful for the simulation of the design. By using nonsynthesizable constructs, the Verilog Design Under Verification (DUV) can be verified to find out functional correctness of the design. Consider the simple Verilog Design of ring counter with inputs as “clk” and the “reset\_n,” counter has four-bit output “q\_out [3:0]” the RTL description of ring counter is shown in Example 9.7.

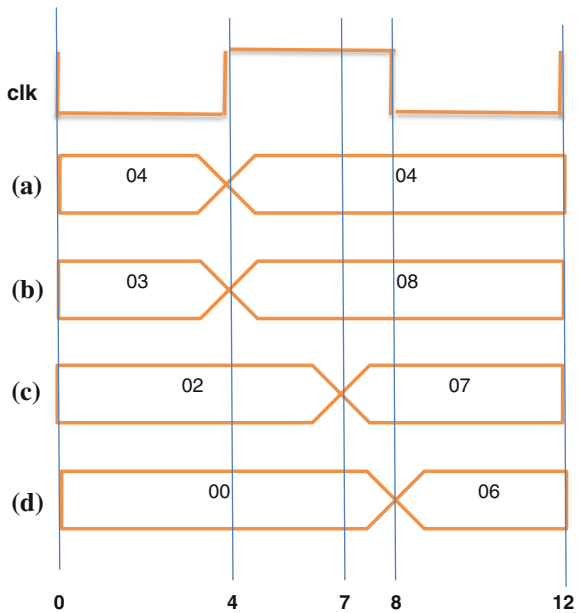
```
always@(posedge clk)
begin
    b<=a+a;
    c<=#3 b+a;

    d<=#1c+a;
end

initial
begin
    a=4;
    b=3;
    c=2;
end
end
```

**Example 9.6** Verilog nonblocking assignment with intra-assignment delay

**Waveform 9.6** Simulation result for the Verilog nonblocking assignment with intra-assignment delay



```
module ring_counter (clk, reset_n, q_out);  
  
input clk;  
  
input reset_n;  
output reg [3:0] q_out;  
  
always@(posedge clk)  
  
begin  
  
if (~reset_n)  
  
    q_out<=4'b1000;  
  
else  
  
begin  
  
    q_out[3] <=q_out[0];  
  
    q_out[2]<=q_out[3];  
  
    q_out[1]<=q_out[2];  
  
    q_out[0]<=q_out[1];  
  
end  
  
end  
  
endmodule
```

**Example 9.7** Four-bit ring counter using Verilog HDL

The testbench for the ring counter is described by Example 9.8 and forces the stimulus to the DUV.

The above testbench generates the results shown in Waveform 9.7.

As discussed above, the basic simulation can be carried out by writing the testbench which can force the stimulus to the design under test. For the lesser complexity FPGA designs, this approach can work. But for large SOC design modules, it is essential to use the sophisticated self-checking testbenches. It is essential for the verification engineer to understand about the creation of the test cases, test plans, and test vectors. Even the best industry practice is making use of the verification architecture by using drivers, monitors and checkers. This discussion is out of scope for the FPGA-based design.

```
//Verilog testbench to force reset_n and clk to ring counter

`timescale 1ns/1ps

module test_ring_counter();

    reg clk_tb, reset_tb;
    wire [3:0] tb_q_out;

    //Instantiation of design under Verification

    ring_counter duv (.clk(clk_tb),

                     .reset_n(tb_reset),

                     .q_out(tb_q_out));

    initial

    begin

        $display("time,\t clk_tb,\t reset_tb,\t tb_q_out");

        $monitor("%g,\t%b,\t%b,\t%b", $time, clk_tb, reset_tb, tb_q_out);

        reset_tb=1'b0; // active low reset stimulus

        #50 reset_tb=1'b1;

        #200 reset_tb=1'b0;

        clk_tb=1'b0; // clk initalization

    end

    always

    begin

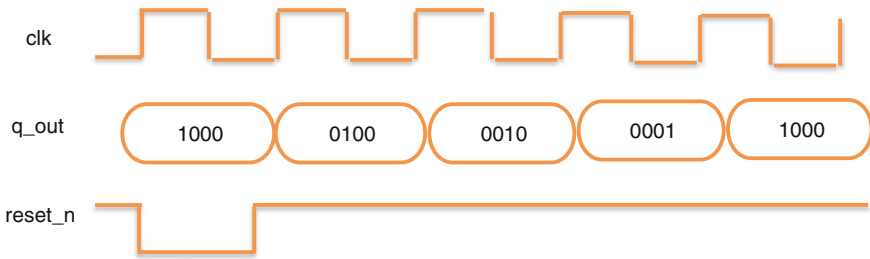
        #10 clk_tb=~clk_tb; //clk generator

    end

    #500 $finish;

endmodule
```

**Example 9.8** Testbench for the Verilog ring counter



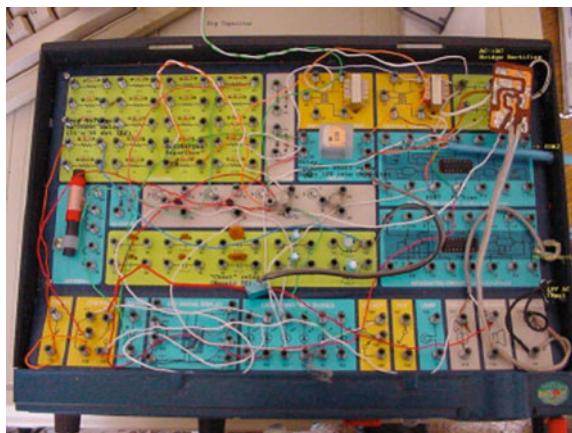
**Waveform 9.7** Simulation result of ring counter

### 9.3 Introduction to PLD

In the past decade, the Programmable logic devices (PLD) market has grown and the demand of the PLDs has increased to realize and prototype the new ideas. The chip which has programmable features and can be programmed is called as PLD. The PLD is also named as filed programmable device (FPD). FPDs are used to implement the digital logic, where the integrated circuit can be configured by the user to realize the different designs. The programming of such integrated circuit is accomplished by using the special programming using the EDA tools.

The first programmable chip introduced in the market was Programmable Read Only Memory (PROM). PROM has a number of address lines and data lines. Address lines are used as logic circuit inputs and data lines are used as logic circuit outputs, as PROM has inefficient architecture and cannot be used to realize the complex digital logic. The device developed during 1970s is PLA which has two levels of logic and is used to realize the small-density logic. After evolution of PLA, the real evolution of programmable logic device took place. After PLA, the SPLD, CPLD, and FPGA evolved during early 1980s. Early programmable logic device is shown in Fig. 9.1.

**Fig. 9.1** Early programmable logic device



**Table 9.2** PLD classification

PLD	SPLD	CPLD	FPGA
Logic cell	PAL or PLA	SPLD	CLB
Density	Few hundred logic gates	Few thousand logic gates	Few lac logic gates
Type	Gate rich logic	Gate reach logic	Flip-flop rich logic
Application	Small density FSM	Moderate gate count FSMs	Complex FSMs

The PLD Classification is shown in Table 9.2.

Following are the key terminologies used to understand the filed programmable devices.

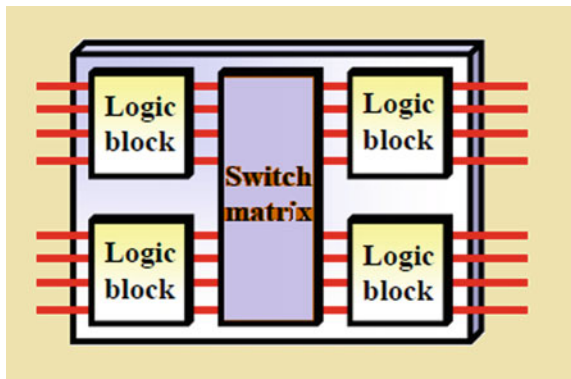
*PAL* A relatively small-density field programmable device (FPD) which has programmable AND plane followed by fixed OR plane is called as Programmable Array Logic (PAL).

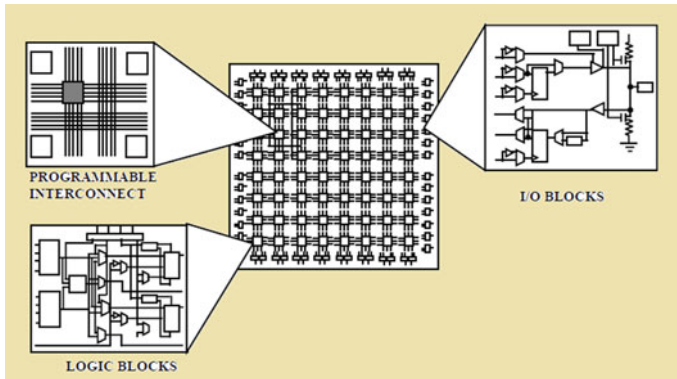
*PLA* A relatively small-density field programmable device (FPD) which has programmable AND plane followed by programmable OR plane is called as Programmable Logic Array (PLA). The PLA structure has two levels of logic and can be available on the full-custom chip.

*SPLD* Any structure which is similar to PAL or PLA is called as simple programmable logic device (SPLD). SPLDs are used to realize small gate count state machines due to the good timing performance.

*CPLD* The structure which consists of multiple SPLD-like blocks on the same chip with interconnection logic is called as *Complex Programmable Logic Device (CPLD)*. CPLD is also called as mega PAL, Super PLA, or Enhanced PLD (EPLD). In the practical scenario, CPLDs are used to realize moderate-density state machines due to improved timing performance as compare to the SPLDs. The CPLD structure is shown in Fig. 9.2.

**Fig. 9.2** Block diagram of CPLD





**Fig. 9.3** Basic FPGA architecture (Source XILINX)

*FPGA* It is the programmable logic which consists of the more number of resources like flip-flops and logic blocks to realize the high-density logic are called as Field Programmable Gate Array (FPGA). FPGA is also called as programmable ASIC and consists of the configurable logic blocks (CLB), IO blocks (IOB), and programmable interconnects. Modern FPGAs even consists of the multipliers, block RAMs, DSP blocks, and processor cores. The FPGA with the key blocks, logic block, IO block, and programmable interconnect is shown in Fig. 9.3.

*Interconnect* The wiring resource in the field programmable device is called as an interconnect.

*Programmable Switch* The switch used to connect one interconnect wire to another or logic block to the interconnect wire is called as programmable switch.

*Configurable logic block (CLB)* The logic block which can be configured for the required combinational and sequential logic functionality is called as CLB. While implementing the logic on the FPGA, the logic is decomposed into small-density logic blocks and mapped on the multiple CLBs.

*Logic density* The amount of logic in the FPGA per unit area is called as logic density.

*Logic capacity* The amount of logic that is mapped into the single-filed programmable device is called as logic capacity. The logic capacity is given in the form of the number of logic gates in the gate array. The logic capacity can be thought in the form of number of two input NAND gates or universal gates.

*Performance* The maximum operating frequency of the field programmable device is the measure of the performance for the sequential logic. For the combinational logic, the longest path in the design decides the performance.

The comparison of the structured ASIC and FPGA design is listed in Table 9.3.

**Table 9.3** Comparison of structured ASIC with FPGA

Selection criteria	Structured ASICs*	EasyPath FPGAs
Time to prototype samples	4–8 weeks	0 weeks
Total time to volume production	12–15 weeks	8 weeks
Vendor NRE/mask costs	\$100K–\$200K	\$75K
Design costs for conversion	\$250K–\$300K	\$0
Additional cost of tools for conversion	\$100K–\$200K	\$0
Unit costs	Low	Low
Risk	High	Low
Flexibility to make changes in system	Inflexible	Flexible
Design conversion from prototype to production	Additional engineering	Conversion free

Source XILINX

\* XILINX Market Analysis

## 9.4 FPGA as Programmable ASIC

Modern FPGAs are named as programmable ASICs and used in various applications which include the ASIC SOC designs and prototyping. FPGA programming includes following types and discussed in this section. The main programming types for any FPGA are

### 9.4.1 SRAM Based FPGA

Most of the FPGAs in the market are based on the SRAM technology. They store the configuration bit-file in the SRAM cells designed using latches. As the SRAM is volatile, they need to be configured at the start. There are two modes for programming and they are Master and Slave. The SRAM memory cell is shown in Fig. 9.4.

In the Master mode, FPGA reads configuration data from the external source and that can be flash.

In the Slave mode, FPGA is configured by using the external master device such as processor. The external configuration interface can be JTAG that is also called as boundary scan.

### 9.4.2 Flash-Based FPGA

In this type of FPGAs, the flash memory is used to store the configuration data. So the primary resource for this FPGA is the flash memory. So these kinds of FPGAs have the less power consumption and they are less tolerant for the radiation effects. In the SRAM-based FPGAs, the internal flash is only used during power-up to load the configuration file. The floating gate transistor used in the flash memory is shown in Fig. 9.5.



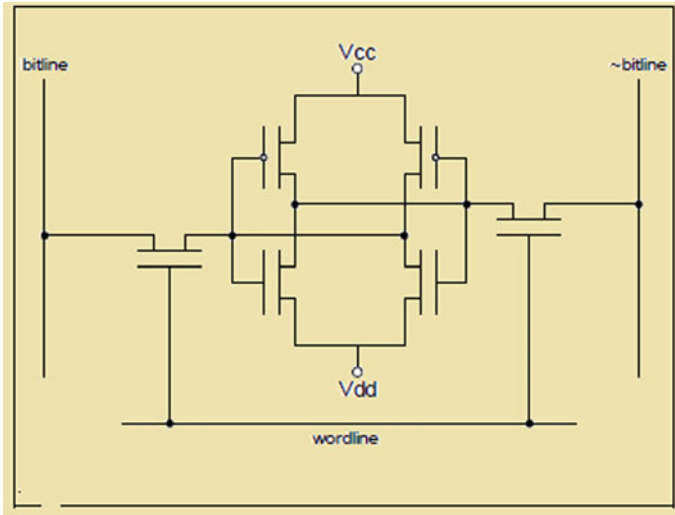
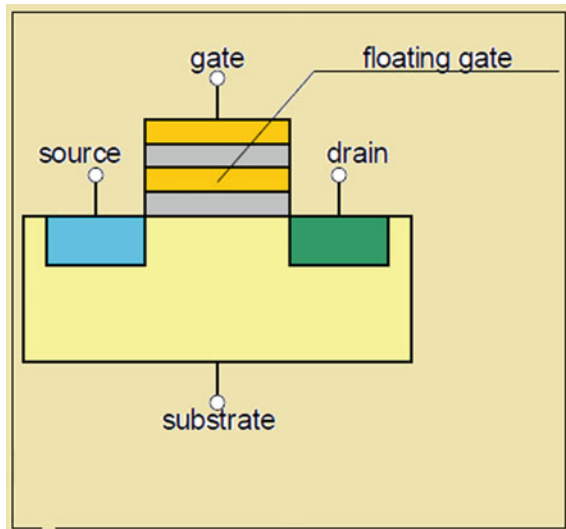


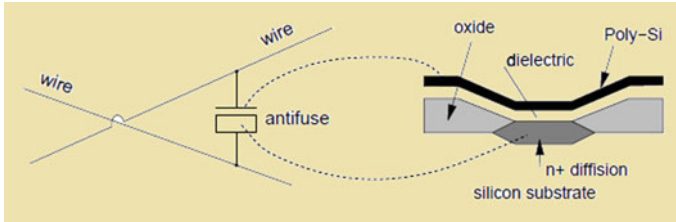
Fig. 9.4 SRAM Cell

Fig. 9.5 Floating gate transistor in flash memory



### 9.4.3 Antifuse FPGAS

These types of FPGAs are used to program only once and they are different as compared to previous two types of FPGAs. Antifuse is opposite to fuse and initially at the start they does not conduct current but can be burned to conduct current.



**Fig. 9.6** Antifuse structure

Once they are programmed, there is no any way to reprogram as burned antifuse cannot be forced to the initial state. It is shown in Fig. 9.6.

### 9.4.4 FPGA Building Blocks

The following are key building blocks in the FPGA architecture and described in this section. The FPGA architecture is shown in Fig. 9.7.

1. *Configurable Logic Block (CLB)* CLB consists of the Look UP Tables (LUTs), multiplexers, and registers. RAM-based LUTs are used to implement the digital logic. CLBs can be programmed to realize wide variety of logic functions. Even CLBs are used to store the data.
2. *Input–Output Block (IOB)* This block is used to control the data flow between the internal logic and IO pins of the device. Each IO is used to support the bidirectional data flow with the tri-state control. There are almost 24 different IO standards which include seven differential special IO high-performance standards. The double data rate registers are also provided with the digital-controlled impedance feature.
3. *Block RAM (BRAM)* They are used to store the large amount of the data and available in the form of dual-port RAM. For example 18-Kbit dual-port block RAM. BRAM can consist of such multiple blocks depending on the device.
4. *Digital clock managers (DCMs)* They are used for clock management and provides fully calibrated digital clock solution. They are used for the uniform clock distribution, delay of clock signals, multiply or divide the clock signals with uniform clock skew.
5. *Multipliers* dedicated multiplier block is used to perform the multiplication of two ‘ $n$ ’ bit digital numbers. Depending on the device the ‘ $n$ ’ can vary. If  $n = 18$  then the dedicated block is used to perform the multiplication of two 18-bit numbers.
6. *DSP blocks* They are embedded DSP blocks used to realize the DSP functions such as filtering, data processing. These blocks are used to improve the overall performance of the FPGA while processing the huge amount of data for the DSP applications.

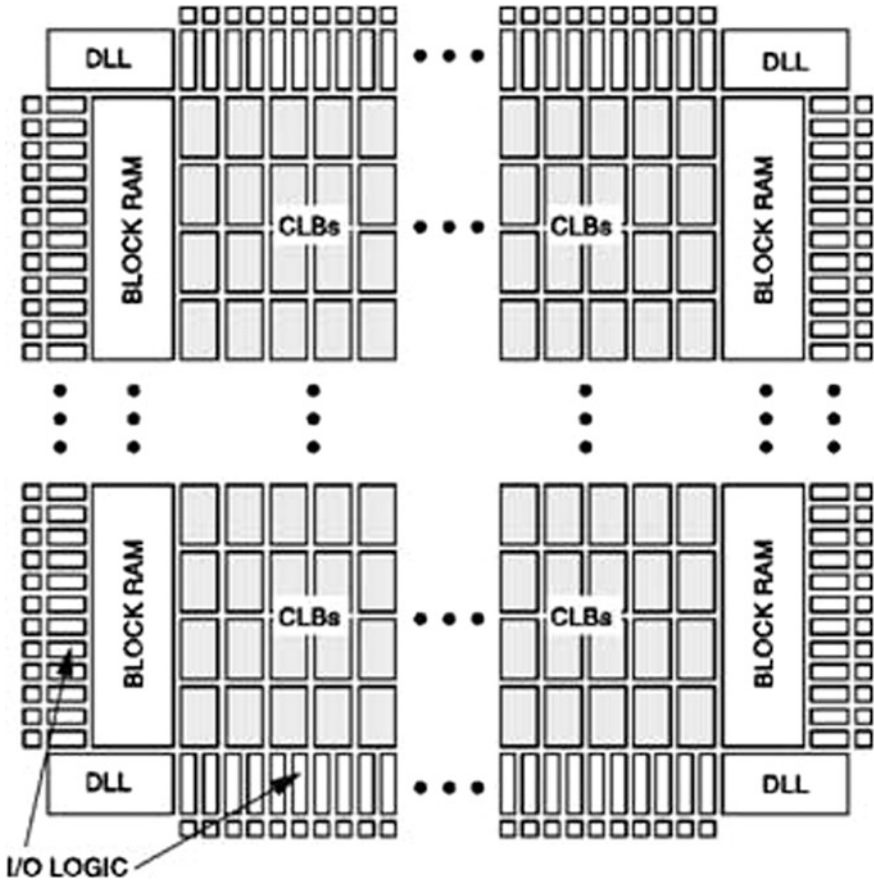


Fig. 9.7 FPGA architecture (Source Xilinx)

### 9.5 FPGA Design Flow

FPGA design flow includes following key steps and described in Fig. 9.8.

1. Design entry
2. Design simulation and synthesis
3. Design implementation
4. Device programming.

The design steps are elaborated in the following section.

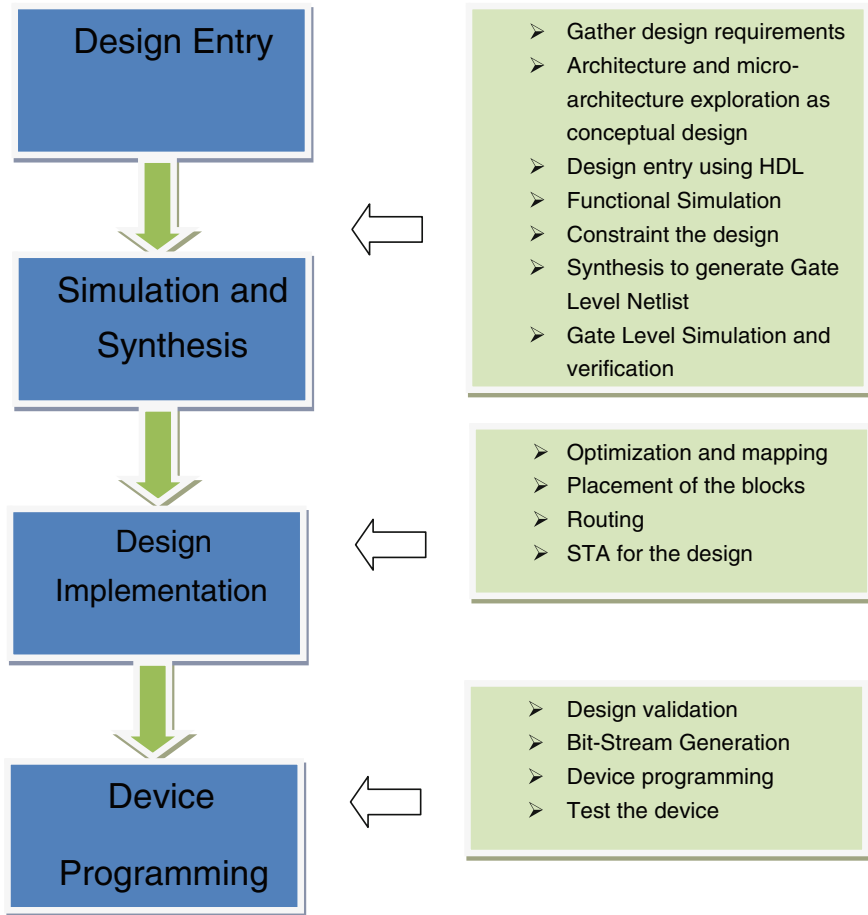


Fig. 9.8 FPGA design flow

### 9.5.1 Design Entry

Before the design entry, the design planning need to be done from the design specifications. The design specifications need to be converted to the architecture and micro-architecture. The design architecture and micro-architecture includes the overall design break-up into small modules to realize the intended functionality. During the architecture design phase the requirement of memory, speed and power need to be estimated. Depending on the requirement the FPGA device need to be chosen for the implementation.

Design entry is done by using either Verilog (.v) or VHDL (.vhd) file. After the design entry the design need to be simulated for the functional correctness of the design. This is called as functional simulation.

### ***9.5.2 Design Simulation and Synthesis***

During functional simulation the set of inputs are applied to the design to check for the functional correctness of the design. Although the timing or area, power problems can crop up during the later design cycle but designer is at least sure about the functionality of the design.

The major goal of the hardware design engineer is to generate the efficient hardware. The synthesis is the process of converting the one level of the design abstraction into the other level. In the logical synthesis, the HDL is converted into the netlist. The netlist is device independent and can be in the standard format like electronic design interchangeable format (EDIF).

### ***9.5.3 Design Implementation***

The design goes through the steps as translate, map and place and route. During the design implementation the EDA tool translates the design into the required format and maps it on to the FPGA depending on the required area. The mapping is performed by the EDA tool by using the actual logic cells or macrocells. During the mapping process the EDA tool uses the macrocells, programmable interconnects and the IO blocks. The special dedicated blocks like multipliers, DSP, and BRAMs are also mapped using vendor tools. The blocks are placed on the predefined geometry inside the FPGA and routed by using the programmable interconnects for the intended functionality. The step is called as place and route.

To check for the design timing performance and whether the constraints are met or not the timing analysis is performed and it is called as post layout STA. During the STA the timing paths are checked with the delays associated with the programmable interconnects. Extracting the RC delays and using them for timing analysis is also called as back annotation.

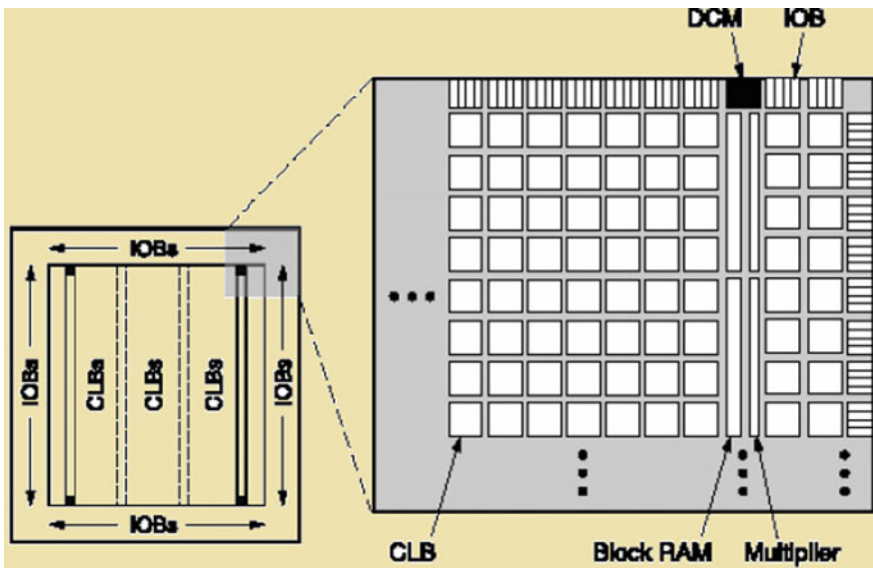
### ***9.5.4 Device Programming***

The FPGA is programmed by using the vendor specific or proprietary bit-stream file. Bit-stream is binary data file that needs to be loaded into the FPGA to execute the particular hardware design.

If the design is targeted with the specific FPGA then the EDA tool generates device utilization summary. Please refer Appendix II for the XILINX Spartan series devices.

### 9.6 Logic Realization Using FPGA

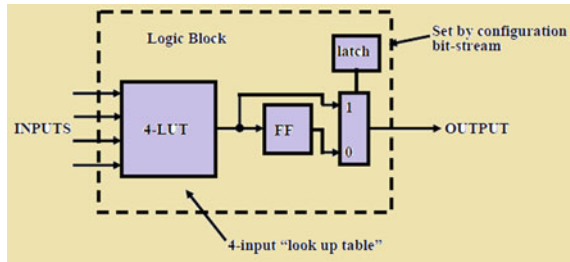
The architecture of modern FPGA consists of the array of CLBs, Block RAMs, Multipliers, DSPs, IOBs, and digital clock managers. Delay-Locked Loop (DLL) is used to distribute the clock with uniform clock skew. The floor plan for the XILINX SPARTAN Series FPGA is shown in the following figure.



#### 9.6.1 Configurable Logic Block

As shown in the following figure basic CLB consists of the LUTs, flip-flop, and multiplexer logic. The configuration data is hold in the latch. The CLB architecture is vendor dependent and can consists of multiple LUTs, flip-flops, multiplexers, and latches. The following Verilog code is realized by using the single four input LUT without register and the output is called as combinatorial output.

**Fig. 9.9** Xilinx basic CLB structure



```
always@(a_in, b_in)
begin
y_out= a_in && b_in;
end
```

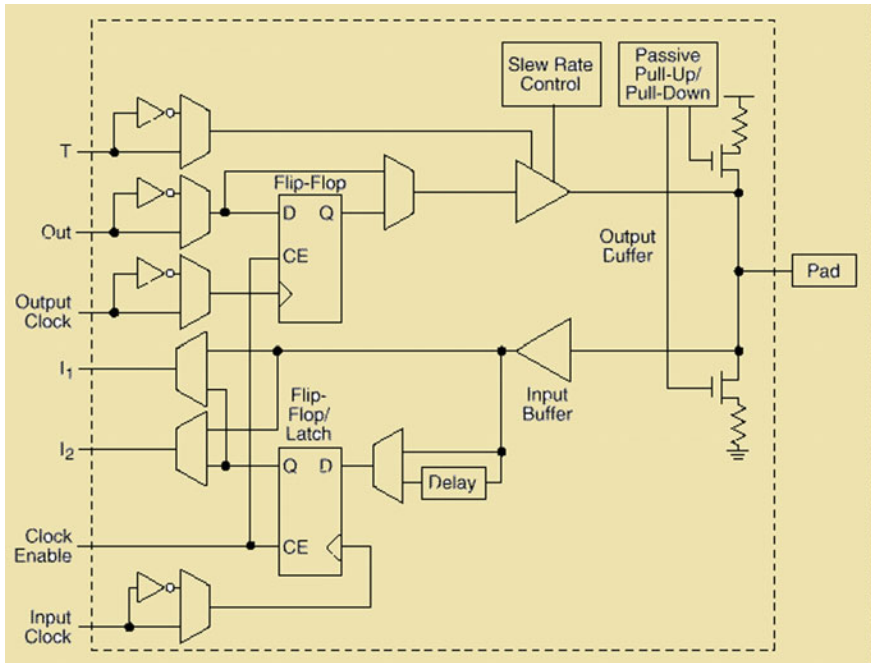
The following Verilog functional block uses single LUT with single register during realization and hence the logic is called as sequential logic.

```
always@(posedge clk)
begin
y_out= a_in && b_in;
end
```

The CLB shown in Fig. 9.9 is also used to implement 16-bit shift register. The LUTs can be cascaded to design the longer size shift register or it can be used for pipelining of the design.

### 9.6.2 Input–Output Block (IOB)

An input–output block is used to establish the interface of the logic with outside world and consists of the number of registers and buffers with the tri-state control mechanism. The block can be used to have a registered input and registered output.



**Fig. 9.10** Xilinx basic IO block

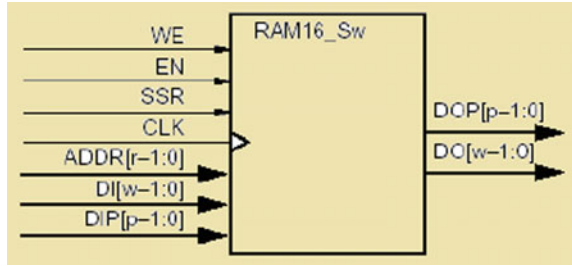
The IOB structure of modern FPGA is complex and can consist of many IO control support which may include DDR, special purpose high-speed interfaces. The basic IO block structure is shown in Fig. 9.10.

### 9.6.3 Block RAM

XILINX Spartan-3 family supports 200 MHz block RAM organized in the four columns and in the form of synchronous configurable 18 kbits blocks. Each Block RAM contains 18,432 bits among them 16 kbits is allocated for the data storage and remaining 2 kbits are allocated for the parity. Block RAM can be used as single port memory or dual-port memory and has independent port access. Each port is synchronous with independent clock, clock enable, and write enable. Read operations are also synchronous in nature and requires the clock enable. The applications of Block RAM is to store the data, FIFO designs, buffers, and stacks and even while designing the complex state machines. Single port RAM is shown in Fig. 9.11.



**Fig. 9.11** Xilinx single port BRAM



### 9.6.4 Digital Clock Manager (DCM) Block

The Xilinx device family uses the delay-locked loop (DLL) and Altera uses the phase-locked loop (PLL) as clock manager. The role of DCM, DLL is to provide complete control over the phase shift, clock skew, and clock frequency. The DCM, DLL supports the following functions.

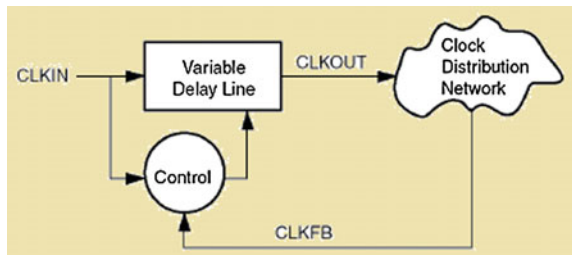
- Phase shifting
- Clock skew elimination
- Frequency synthesis.

The DCM consists of the variable delay line and clock distribution network and basic block diagram is shown in Fig. 9.12.

### 9.6.5 Multiplier Block

All Spartan3 FPGA has two 18-bit inputs and it generates 36-bit output. The multiplier is embedded block and each device has 4–104 embedded multiplier blocks. The main advantage of embedded multiplier is that it requires the lesser power as compared to the CLB-based multipliers. They are used to implement the fast arithmetic functions with minimum use of the general purpose resources. Cascading of multiplier using the routing resources is possible and following figure shows the multiplier configured as 22-bit multiplied by 16-bit to generate the 38-bit

**Fig. 9.12** Xilinx basic DLL block



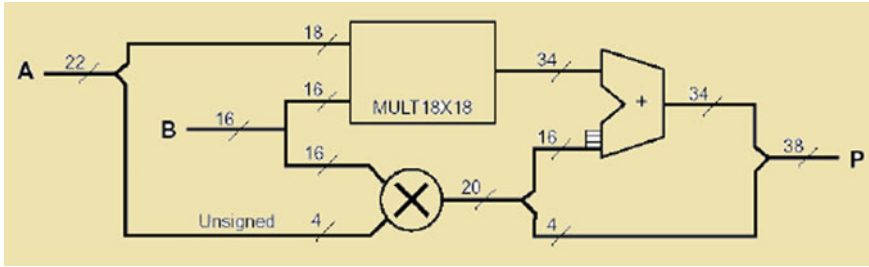


Fig. 9.13 Xilinx basic multiplier block

product. The multiplier can be used for the signed or unsigned number multiplication. The multipliers are extensively used in the DSP applications. The basic block is shown in Fig. 9.13.

## 9.7 Design Guidelines for FPGA-Based Designs

The RTL Verilog codes discussed in Chaps. 1–8 can be targeted on the suitable FPGA design by using the design guidelines for the FPGA. Following are the few design guidelines need to be followed while designing by using FPGAs.

### 9.7.1 Verilog Coding Guidelines

Guidelines for using Verilog to implement efficient RTL are listed in this section and it is always recommended to use these guidelines during RTL design phase. Among these, few guidelines are mainly described with reference to Verilog Stratified Event Queue discussed in Chaps. 4 and 6.

#### 9.7.1.1 Blocking Versus Nonblocking Assignments: (Please Refer Chaps. 4 and 6)

- I. It is recommended to use **blocking assignments** while modeling the **combinational design**.
- II. It is recommended to use **nonblocking assignments** while modeling **sequential design**.
- III. It is recommended to use the **nonblocking** assignments while modeling the **latches**. While implementing RTL design, it is essential to overcome the potential unintentional latches. Unintentional latches are inferred due to missing **else** or due to incomplete **case** conditions.

- IV. It is recommended to use the **nonblocking** assignments while modeling both **sequential and combinational logic**.
- V. It is recommended, **not to mix the blocking and nonblocking assignments** in the same always block.

### 9.7.1.2 Priority Versus Parallel Logic

- I. It is recommended to use **if-else** statement for designing priority logic. Priority encoder or priority interrupt control logic can be modeled by using the **nested if-else** statements. It is recommended to use **case** statement for designing parallel logic. Priority logic generates the longer combinational path due to **nested if-else statements**, so it is always recommended to use **case** statement to generate parallel logic.

## 9.7.2 FSM Guidelines

- I. Binary encoding techniques are efficient for a design having 16 or fewer states. As number of states increases the next state combinational logic performs slower operation.
- II. One-hot encoding technique is efficient and reliable as compared to the binary encoding due to glitch free behavior. One-hot encoding requires low density next state logic and useful in design of larger FSM blocks. But the main drawback of one-hot encoding is that it uses more registers.
- III. While designing FSM, designer need to take care of following key points
  - a. Do not leave any undefined states. Initialize the unused states to reset value or use the default statements.
  - b. Do not implement the FSM with combination of registers and latches. Avoid the unintentional latches in the FSM design to improve the reliability.
  - c. Model the FSM blocks by using **case** statements to infer the parallel logic.
  - d. Separate the next state, output combinational logic and state register logic in different always block to improve the speed of FSM and for better synthesis results.
  - e. Register FSM output as it preserves the hierarchy.
  - f. Use the look ahead mealy machines for better design performance.

### 9.7.3 *Combinational Design and Combinational Loops*

- I. It is recommended to use **continuous assignment** statement for design of **combinational logic**.
- II. While designing the combinational logic it is essential to avoid the combinational loops. Combinational loop causes instability and unreliability in digital designs as it violates the synchronous design concepts due to infinite looping. The combinational loop generates the oscillatory output and the period of the oscillatory output signal is mainly dependent on the delay introduced by combinational logic in the feedback path.

### 9.7.4 *Grouping the Terms*

- I. Use the signal grouping to improve the performance of FPGA-based design. For example, if the expression  $q = (x + y + z + w)$  is implemented on FPGA then hardware inference is cascade structure but with grouping by using expression  $q = (x + y) + (z + w)$  the hardware inference is parallel structure. Due to grouping the timing performance of design is improved.

### 9.7.5 *Assignments*

- I. It is recommended **not to make the assignments to same variable from multiple always block**. It gives error as multiple drivers to the same net or wire.
- II. It is recommended **not to make assignments with #0 delay [1]**.

### 9.7.6 *Simulation and Synthesis Mismatch*

Most of the synthesis tools ignore the sensitivity list of combinational procedural block but simulator executes the procedural block, only when there is event on one of the signal in the sensitivity list parameters. Due to incomplete sensitivity list it creates the simulation synthesis mismatch.

### 9.7.7 *Post-synthesis Verification*

It is highly recommended to perform the post-synthesis verification for the FPGA-based design. Post-synthesis verification with the SDF assures the correct behavior of the gate level netlist. There should not be mismatch between the functional verification of the design and the post-synthesis verification.

### 9.7.8 *Guidelines for Area Optimization*

FPGAs have finite resources so it is recommended to follow the design guidelines to optimize the area. The area optimization techniques are: resource sharing, logic duplication. [Note: Many times it has been observed that, logic duplication can even increase area and the use of logic duplication technique is dependent on the design scenarios.]

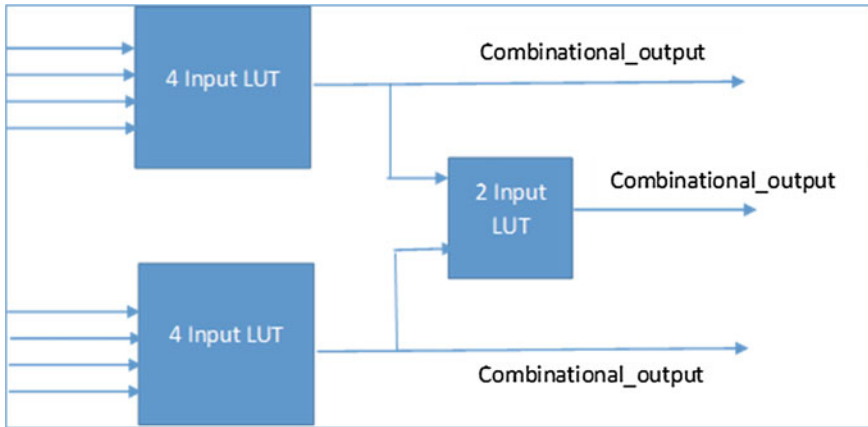
#### 9.7.8.1 **Resource Sharing**

Always it is observed that, adders consume more area as compared to multiplexers. The resource sharing is a powerful technique to share the common resources to minimize the area. It is essential for the FPGA designer to consider resource sharing of arithmetic operators used in the same hierarchy. The resource sharing is one of the powerful area minimization techniques. But it is recommended that not to share resources from different modules or from different hierarchy. Resources can be shared from the same module or from the same hierarchies.

#### 9.7.8.2 **Logic Duplication**

It is the powerful technique to reduce the net delay by enabling the placement tool to place the replicated logic in various areas of die [2]. The major drawback of this technique is that it increases the area of the design while replicating the register or sequential logic.

As per as FPGA area minimization is concerned, logic duplication can act as a very efficient tool but depends on the design specific scenarios. Consider example of implementing 8:256 decoder using single **case** statement. If FPGA architecture has logic block with two, four input LUTs and output generation LUT as shown in Fig. 9.14 [3] then to implement the single output it uses three LUTs. So for 256-bit output 768 LUTs are utilized. By splitting case statement to implement two 4:16 decoders, logic duplication can be achieved. By using logic duplication, if two 4:16 decoders are used with 256 AND gate array then the overall device utilization is just 288 LUTs for implementation of 8:256 decoder and it reduces the device utilization



**Fig. 9.14** Xilinx basic LUT structure

by around 480 LUTs. That is very huge reduction in the overall area. For the 8:256 decoder the logic duplication is accomplished by using the four input LUTs and two input LUTs; the structure of logic block is shown in Fig. 9.14 [3].

### 9.7.9 Guidelines for Clock

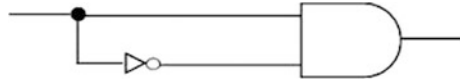
The performance and reliability of an FPGA-based design are based upon the clocking schemes. For the FPGA-based design and implementation it is recommended that

- a. Use single global clock.
- b. Avoid use of gated clocks.
- c. Avoid mixed use of positive and negative edge triggered flip-flops.
- d. Avoid use of internally generated clock signals.
- e. Avoid ripple counters and asynchronous clock division.

It is recommended by most of the FPGA vendors that do not use the internal generated clocks as it causes the functional and timing issues in the design. If internal generated clocks are required in the design then use DLL [3] or PLL [2] to generate the clocks. The internal generated clocks by using combinational logic are prone to glitches and it creates the functionality issues in the design. Due to the combinational delays it create the timing issues in the FPGA designs. The major problem for using the internal generated clocks is the issue during synthesis and timing analysis. Xilinx [3] provides the library component global clock buffers BUFGCTL [3] and BUFGMUX [3] to generate internal clocks.

To avoid glitches it is recommended to register the output of the internal generated clocks. It is recommended to use the clock generation logic. For low power

**Fig. 9.15** Asynchronous pulse generator



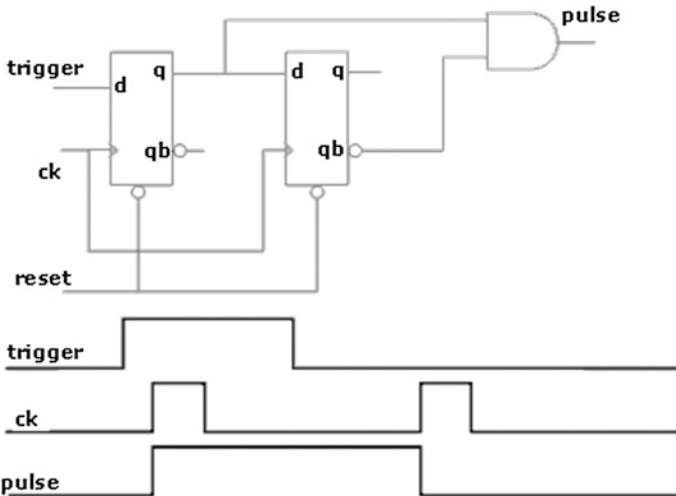
designs it is essential to use the clock gating but it is prone to glitches. So it is recommended to use the clock gating cells for low power FPGA-based design.

It is recommended not to use the asynchronous pulse generator circuit. Figure 9.15 represents the asynchronous way of pulse generation. This technique should be avoided as it is prone to glitches and very difficult to synthesize and place and route. Depending on the pulse width requirement replace the inverter shown in figure; by chain of odd number of inverters.

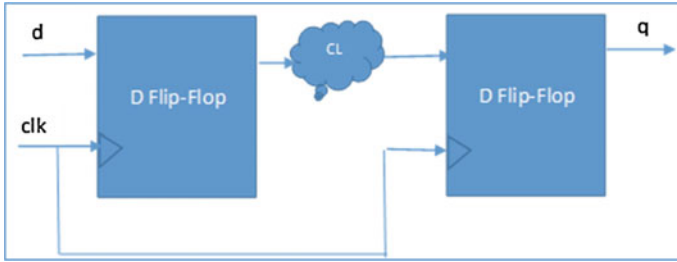
Figure 9.16 represents the recommended pulse generator where the pulse width is dependent on the clock period. It is recommended to use two-level synchronizer at the input of pulse generator to avoid the metastability issues.

### 9.7.10 Synchronous Versus Asynchronous Designs

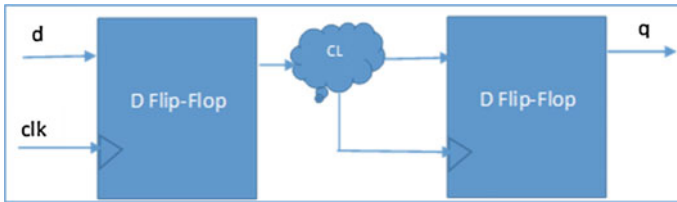
In synchronous design the data input is sampled on every active edge of clock and clock signal controls the activities of inputs and outputs. Figure 9.17 represents the synchronous design where the combinational logic (CL) drives the data to the input of flip-flop. For the proper operation of the design, it is essential that the data input should be stable for at least setup time of register and it should be stable for at least hold time of register. The propagation delay of combinational logic limits the



**Fig. 9.16** Synchronous pulse generator



**Fig. 9.17** Synchronous logic



**Fig. 9.18** Asynchronous logic

operating frequency of the design. To meet the timing requirement it is essential to have synchronous relationship of all inputs and combinational inputs with the clock signal of the flip-flop. Use the pipelining feature to improve the performance of synchronous design. As FPGA is register rich, logic pipelining is used for improving the speed of the design at the cost of latency.

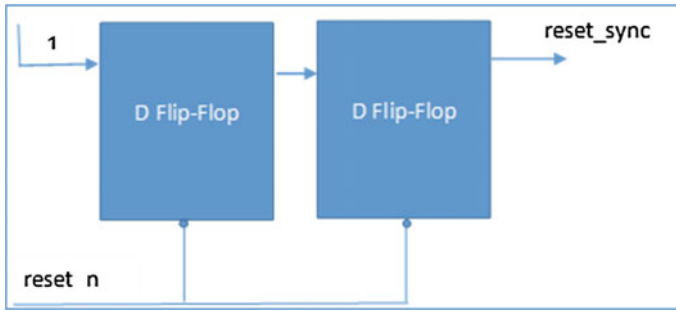
On the other hand, an asynchronous design does not have common clock (Example Ripple counters) and are prone to glitches or spikes. It is very difficult to model the timing of asynchronous design by using timing constraints. Many times an asynchronous design generates the glitches or short time duration pulses shorter than the clock period. If the glitches are passed through the combinational logic then the output leads to an incorrect value. Figure 9.18 describes an asynchronous logic prone to glitches.

Many times it has been observed that an asynchronous logic reduces the device resources but prone to hazards. So, it is recommended to use the synchronous logic while implementing the sequential design. Synchronous logic always makes STA easy [4].

### 9.7.11 Guidelines for Use of Reset

Resets are classified as synchronous and asynchronous resets. Asynchronous resets are easy to implement as they do not depend on the clock. But STA becomes





**Fig. 9.19** Reset generation logic

difficult and complex while using asynchronous resets. At the same time automatic insertion of the test structure is difficult.

On the other hand, synchronous resets are difficult to implement as it requires more resources and they are dependent on the clock. Synchronous resets slowdown the design performance. It is recommended that FPGA designer should avoid internally generated conditional resets [2, 3].

It has been observed during FPGA-based designs that, reset deassertion circuit is required while using asynchronous reset. If reset signal is deasserted and if does not pass the setup and hold timing check then flip-flop goes into metastable state and it can lead to potential functional issues in the design [5].

It is recommended to use the synchronized asynchronous resets. That is asynchronously asserted and synchronously deasserted. Figure 9.19 is the recommended representation of asynchronous active low reset (`reset_n`) passing through the two-level synchronizer.

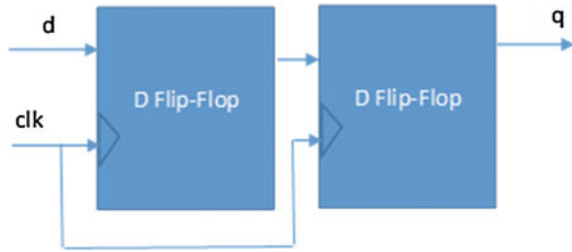
For very large density or complex FPGA-based designs with multiple hierarchies, it is essential to use the Linting tool which can provide proper information about the reset and clock trees [2, 3].

### 9.7.12 Guidelines for CDC

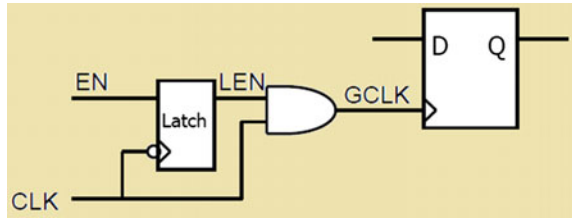
It is impossible to verify the clock domain crossing (CDC) by using functional verification and even it is impossible to verify CDC by using timing analysis tool due to asynchronous nature of clock path. The major problem encountered in CDC is functionality failure due to metastability. To avoid metastability it is recommended to use the dual or three-stage synchronizers while transferring signals from one clock domain to another.

Linting tools are used to ensure the use of synchronizer chain on the clock domain crossing paths. Use two or three-level synchronizer shown in Fig. 9.20 to transfer the signals from one clock domain to another. This will avoid metastability in the design.

**Fig. 9.20** Two-level synchronizer



**Fig. 9.21** Low power clock gating cell



### 9.7.13 Guidelines for Low Power Design

Reducing the power for many applications is very critical and due to complexity of designs only use of power efficient FPGA devices or architecture is not sufficient. It is essential for designer to understand the features of EDA tools to optimize the dynamic power. The recommendation by many FPGA vendors is to reduce the switching activity in the sequential logic and clock routing [2, 3]. For the low power design, it is recommended to use the gated clocks or the low power clock gating cells. Dynamic power of a cell is dependent on voltage, load capacitance, and on clock frequency. Due to switching at the clock input it has been observed that the dynamic power increases. So to reduce dynamic power it is recommended to use clock gating cells. Figure 9.21 shows the clock gating cell.

### 9.7.14 Guidelines for Use of Vendor-Specific IP Blocks

It is always recommended by the FPGA vendor to have the brief and detail understanding of the FPGA device and the architecture of FPGA device. It is recommended to use the vendor-specific design and coding guidelines to improve the performance of design. It is highly recommended to encrypt the IP by using proper security standards.

During synthesis phase it is recommended to infer the micro-functions such as multipliers, shift registers, memories, and DSP blocks to ensure the optimal results [2, 3].

For the better performance, it is recommended to use the proper timing constraints and analyze the timing constraints by using the timing analyzer [3]. It is even recommended to use the proper place and route effort level while implementing the design [3]. The place and route effort level allows the EDA tool to use the proper algorithm to improve the design performance and even it improves the design placement. It is also recommended to use the proper IOB resources and proper speed grade during design implementation stage [3]. While using the synchronous interface, it is recommended to use the single clock synchronous RAM (read and write in the same clock domain) and while using asynchronous interfaces use the dual-port RAM [2].

## 9.8 Summary

Following are key points to summarize this chapter.

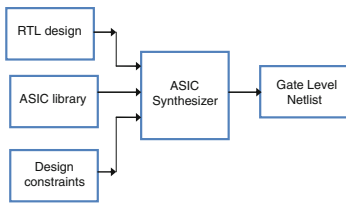
1. Procedural block “initial” is nonsynthesizable construct.
2. Blocking assignment with inter-assignment delay, delays both the update and evaluation.
3. Blocking assignment with the intra-assignment delay, delays the update but not the evaluation.
4. NonBlocking assignment with inter-assignment delay, delays both the update and evaluation.
5. NonBlocking assignment with the intra-assignment delay, delays the update but not the evaluation.
6. PLDs are classified into three main categories and are SPLD, CPLD, and FPGAs.
7. PAL, PLA are also called as SPLDs and used to realize small gate count designs.
8. CPLDs are moderate-density FPDs and are used to design small gate count FSMs due to good timing performance.
9. FPGAs are used to design the complex gate count FSMs and are called as flip-flop rich logic.
10. Modern FPGA consists of the dedicated multipliers, DSP blocks with the processor cores.
11. FPGA designer needs to use the design guidelines while using the FPGAs.

## References

1. IEEE standard. <http://standards.ieee.org/getieee/1800/download/1800-2012.pdf>, [www.ieee.org](http://www.ieee.org)
2. Altera "Quartus II Handbook". [www.altera.com](http://www.altera.com)
3. Xilinx ISE simulation and synthesis guide. [www.xilinx.com/support/documentation/sw\\_manuals/xilinx14.../sim.pdf](http://www.xilinx.com/support/documentation/sw_manuals/xilinx14.../sim.pdf), [www.xilinx.com](http://www.xilinx.com)
4. Wolf W (2005) FPGA based system design. Prentice Hall
5. Altera Quartus II documentation. [www.altera.com/literature/hb/qts/quartusii\\_handbook.pdf](http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf), [www.altera.com](http://www.altera.com)

# Chapter 10

## ASIC RTL Synthesis



ASIC is an Application Specific Integrated Circuit and designed for the specific applications. For ASIC design engineer it is required to have god understanding of the ASIC synthesis and optimization. The chapter discusses about the RTL synthesis and optimization techniques.

**Abstract** Application Specific Integrated Circuit (ASIC) is designed for the specific purpose. The ASIC design flow can be used to design the full-custom or semi-custom designs. This chapter discusses about the different types of ASIC, ASIC design flow key steps, and the RTL synthesis. The design optimization techniques and the Synopsys Design Compiler commands are covered in this chapter with relevant examples. This chapter also discusses about key Verilog RTL modifications to reduce the compiler time during synthesis.

**Keywords** ASIC · FPGA · STA · Data path · Control path · Library · Link library · Target library · Search path · Technology library · Design constraints · Optimization constraints · Resource allocation · Structuring · Partitioning · Clock definitions · Skew definitions · Input delay · Output delay · Read design · Check design · Analyze · Elaborate · Compile · Map efforts

## 10.1 What Is ASIC?

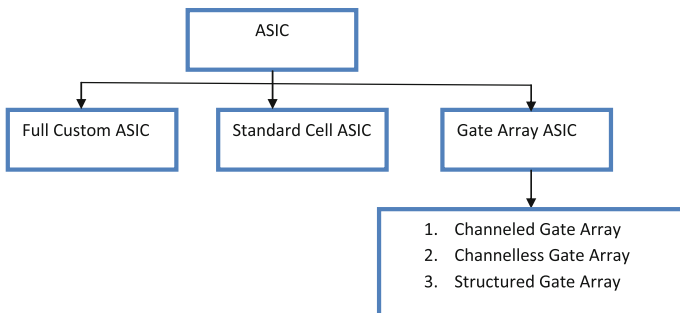
ASIC stands for Application Specific Integrated Circuit. Integrated circuits are made up of silicon wafer and each silicon wafer consists of thousands of die. If any integrated circuit is designed for specific application then it is called as an ASIC. The examples are chip designed for the car controller, chip designed for satellite communication, interfacing chips to establish communication between the CPU and memory. The microprocessors, memories are general-purpose integrated circuits and are not treated as an ASIC. Following are main types of ASIC and shown in Fig. 10.1.

### 10.1.1 Full-Custom ASIC

In such type of ASIC the design starts from the scratch. The ASIC design engineers create the ASIC logic cells and layout required for all the logic. The analog and digital design can be implemented by using full-custom ASICs. In such type of ASICs predefined standard cells or gates are not used to describe the functionality of the design.

### 10.1.2 Standard Cell ASIC

In such type of ASIC, the designer uses the predefined logic cells which are also called as standard cells. Few of the standard cells are logic gates, MUX, flip-flops, or latches. These standard cells are predefined and pretested so designer saves the design time and money and there is less risk while using these standard cells. These types of ASIC designs are flexible like full-custom ASIC designs but reduce overall risk. The standard cell libraries are designed by using full-custom design flow but in semi-custom design already designed libraries are used.



**Fig. 10.1** Type of ASICs

### 10.1.3 Gate Array ASIC

In such type of ASICs, the array which consists of number of transistors is pre-defined on the silicon wafer. The array is also called as base or basic array and the transistor cell is called as basic cell or base cell. The interconnect between the cell and the inside structure of the cell is customized and hence improves the programmability. The types of these ASICs are as follows:

- a. Channeled Gate Array
- b. Channelless Gate Array
- c. Structured Gate Array.

While designing the ASIC following are key objectives need to be considered:

1. *Speed of an ASIC* Whether the ASIC is working at the desired high speed or not.
2. *Area of an ASIC* What is the maximum area of an ASIC?
3. *Power of an ASIC* What is the leakage and dynamic power dissipation in the best case and worst case scenarios?
4. *Time to Market of an ASIC* What is the time to market for an ASIC?

## 10.2 ASIC Design Flow

To design an ASIC, the designer needs to have in-depth understanding of the key steps from specification to the layout. These key steps are used to define the design flow. Figure 10.2 shows the simple ASIC design flow with key steps used in the design cycle.

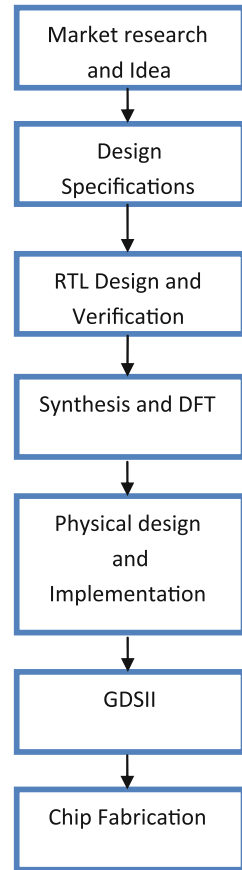
As shown in the above figure, an ASIC design flow consists of the key design steps and can be treated as design milestones. Every ASIC design starts with the basic idea and the idea to develop chip functionality is an outcome of the in-depth market research. After the idea is freeze for the design functionality, the actual ASIC design implementation cycle starts with the specification definitions. The following section describes the key steps in the ASIC design flow.

### 10.2.1 Design Specification

The input to the design specification is the data collected through the market research or the data for the feasible ideas. Following are the key points need to be described in the design specification document:

- a. Functionality of the design. That is what the chip exactly does?
- b. Design goals and constraints for the design
- c. Performance constraints like the speed, power, and area for the said design

**Fig. 10.2** Basic ASIC design flow



- d. Technology constraints like the physical dimensions, space and size for the cell level design
- e. Fabrication techniques for the ASIC design
- f. Vendor-dependent constraints and third-party IPs
- g. Memories and macros used for the design
- h. The data rate and the interface definitions for the design
- i. Packaging information and the testing or verification planning for the design
- j. Risk and dependability and time to market for the design.

The above specifications are described in the form of block diagrams and this phase is called as architecture level design. As discussed in the previous section, the architecture consists of the block level representation of an ASIC design. For example if 16-bit processor needs to be designed then the architecture can consists of ALU, control logic, instruction decoder and encoder, interrupt logic, serial IO controller, BUS arbitration logic, counters and pointer logic. All the mentioned blocks are interconnected together to form the desired architecture required for the



specific application. The chip architect designs the multiple architectures and the best possible architecture for an ASIC is frozen depending on the requirement of speed, power, and resources. This architecture document is used in the ASIC design cycle to describe the functionality of each and every block.

After the architecture for an ASIC is frozen the architecture blocks are described in the form of the small blocks with the interface and logic details and this is called as microarchitecture of the design. The microarchitecture for every functional block can be represented for the intended design functionality using the logic elements. The chip architect with good amount of industrial experience can design the viable and feasible microarchitecture by understanding the functional, timing, and power requirements. Most of the ASIC microarchitecture uses the low power definitions, the DFT friendly design details, the timing details for the interfaces, and the area details. In the microarchitecture the software and hardware design partitioning is defined with the technology-dependent component details.

### ***10.2.2 RTL Design and Verification***

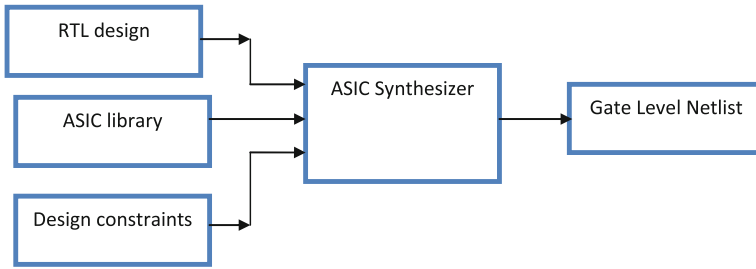
The key milestone is to describe the functionality of an ASIC using synthesizable HDL either Verilog or VHDL is called as the RTL design. RTL stands for the Register Transfer Level and can be efficiently described by using HDL. RTL design engineer uses the microarchitecture document as an input to describe the design functionality using either Verilog or VHDL. The goal of the RTL design team is to describe the design functionality to realize the synthesized logic. The functionalities can be ALU implementations, pipelined features, state machine coding, data transfer modules, memories, etc.

The RTL code is given as an input to the verification engine and the verification team uses this for early detection of the bugs. The RTL verification for any ASIC is an important milestone and it improves the overall coverage for the design. Most of the ASIC design flow uses the verification methodologies and languages like Verilog or system Verilog for the verification of the design.

### ***10.2.3 ASIC Synthesis***

Once the RTL verification is completed and the coverage goals are met the RTL is converted into the optimized gate level netlist. The process of converting an RTL design into the gate level netlist is called as synthesis. The EDA tool uses the Verilog or VHDL RTL, design constraints, and the standard cell library as an input and generates the gate level netlist as an output. Figure 10.3 shows the synthesizer inputs and outputs.

The popular synthesis tools in the industry are Synopsys Design Compiler, Cadence RTL Compiler etc. The synthesis tool considers time, power, and



**Fig. 10.3** ASIC synthesizer inputs and output

testability as the major important factors to generate the gate level netlist. Synthesizer tries to meet the constraints by calculating the cost of various implementations. The gate level netlist is the structural description with only standard cells. The gate level netlist is verified for the functional correctness of the design and this is called as gate level verification.

After successful verification of the RTL design the design need to be checked for the timing violations. This process is called as pre layout STA. During this milestone the goal of ASIC engineer is to find the timing violations for the design. During this phase STA is performed without considering the parasitic (RC) effect. The objective is to fix the setup time violations in the design and to improve the overall performance of the design. In most of the ASICs the hold time violations for various timing paths are fixed after CTS and routing.

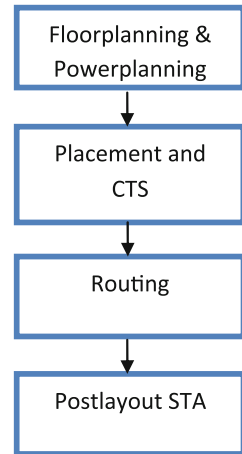
Before physical implementation of an ASIC design, the gate level netlist is given as an input to DFT (Design For Testability) tool. The objective is to find out the various design faults. As discussed above the RTL should be DFT friendly for the quicker scan chain insertions and to find out the overall fault coverage for the design. The DFT techniques and processes are included in Appendix III.

### ***10.2.4 Physical Design and Implementation***

The next milestone in the ASIC flow is physical design and implementation. In this phase the gate level netlist is converted into geometric representations. The geometric representation can be treated as the layout of the design. The discussion on the physical design is out of scope and readers are requested to refer the physical design and synthesis books. The basic physical design flow is shown in Fig. 10.4 and it consists of the following key steps.

The gate level netlist is given to the physical implementation tool to generate the layout. The key steps in the physical design implementation are floor planning, power planning, placement of standard cells and macros, clock tree synthesis, and routing. The design is basically converted from the gate level abstraction to the

**Fig. 10.4** Physical design key steps



switch level abstraction by using CMOS cells. The netlist generated is given to the STA tool to fix the timing violations and this process is called as post layout STA.

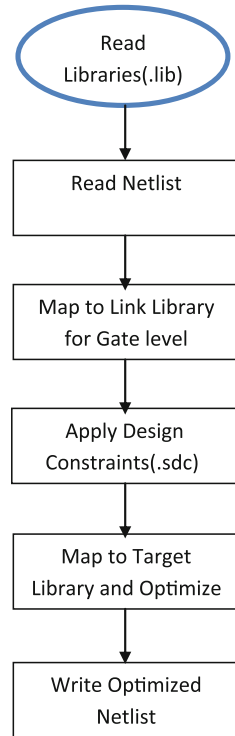
The physical design and implementation tool uses the design rule library to produce the GDSII file. The design rule library consists of the guidelines based on the fabrication processes. GDSII file is used by the foundry to fabricate the integrated circuit. The industry leading tool for the physical design and implementation is IC Compiler from Synopsys or Encounter from Cadence.

The physical verification needs to be performed to verify the intended design functionality and to make sure that the layout is designed according to the rules! After the physical verification and timing analysis the layout is ready for the fabrication. In this phase the layout data is converted into the photo lithography masks. After the fabrication process the wafer is diced into the various individual chips and packaged as well as tested.

### 10.3 ASIC Synthesis Using Design Compiler

This section only focuses on the logical synthesis using the Design Compiler to perform the RTL to gate level synthesis. As discussed above, the RTL synthesis tool uses the RTL design either Verilog (.v) or VHDL (.vhd) files, the design constraints (.sdc) and library (.lib) as an input and generates the optimized gate level netlist using standard cells available in the library. The gate level netlist is technology dependent and can change if process node varies. Depending on the functionality the gate level netlist for the 40 nm can be different as compared to gate level netlist generated for the lower process nodes like 20 or 14 nm process node. ASIC synthesis tool perform internally few steps to generate the gate level netlist. The key steps for the ASIC synthesis are translate, map, and optimize. The key

**Fig. 10.5** ASIC synthesizer key steps



steps for the FPGA synthesis are translate, optimize and map. Figure 10.5 gives the brief information about the ASIC synthesis steps to generate the gate level netlist.

1. **Read Library:** To perform the RTL synthesis, the synthesizer reads the designWare libraries, technology libraries, and symbol libraries. The designWare library consists of the complex cells like adders, comparators, multipliers, etc. The technology library consists of the logic gates, flip-flops, latches, etc. While synthesizing the synthesizer algorithms automatically determine when to use the technology library cells and when to use the designWare library components. These library cells are used efficiently to generate the gate level netlist.
2. The next step is to read the RTL description described by using either Verilog or VHDL.
3. The synthesis tool after reading the libraries and the HDL performs many required steps like optimization, conversion to unoptimized Boolean logic, technology-independent optimization and finally maps the logic using the technology library, the library is also called as target library. The above process is called as linking the logic to the desired target library.
4. The synthesizer uses the design constraints like area, speed, and power while optimizing the design using the standard cells in the target library. So basically

link library can be IO library, cell library, or macro library and used to link the design, target library is used while optimizing the design.

5. For efficient RTL coding it is required that, RTL design engineer should have good understanding of the target standard cell library. After the design is optimized then the design is ready for the DFT, the goal is to detect early faults in the design. During RTL design stage only, the DFT friendly RTL need to be described to enable quick scan insertions and testing for various faults in the design.
6. The optimized netlist can be in the Verilog (.v) format or in the database (.ddc) format and used by the placement and routing tool. Based on the routing the back-annotation can be performed with actual routing delays for accurate timing analysis. If timing goals are not met then the design can be resynthesized to meet the timing goals.

## 10.4 ASIC Synthesis Guidelines

After invoking the Synopsys Design Compiler it reads the startup file from the current working directory. The startup file is **.synopsys\_dc.setup**. There should be two startup files: one should be in the current working directory and another should be in the root directory where the Design Compiler is installed. To use the tool the following important parameters need to be setup.

1. *search\_path* This parameter is used to search for the synthesis technology library for reference during synthesis.
2. *target\_library* This parameter is used by the synthesizer while mapping the logic cells during synthesis. The target library consists of the logic cells.
3. *symbol\_library* All the logic cells have symbolical representation used for the visualization after synthesis. This parameter is used to pint the library that contains the visual information for the logic cells present in the technology synthesis library.
4. *link\_library* The tool uses the cells from the *target\_library* for mapping the desired functionality, this parameter is used to pint the reference of the logic gates in the synthesis technology library.

The above four parameters for **.synopsys\_dc.setup** are described by using following [1]

```
set search_path “./synopsys/libraries/syn/cell_library/syn”
set target_library “tcbn65lpwc.db, tcbn65lpsc.db”
set link_library “$target_library $symbol_library”
set symbol_library “standard.sldb dw_foundation.sldb”
```

**Table 10.1** Design objects used by synthesizer

Design object	Description
Cell	Cell is also called as instance. The instantiated name of the sub-design is called as cell
Reference	It is original design to which cell or instance refers. For example instantiated sub-design must refer to the design which consists of the functional description of the sub-design
Ports	The primary inputs and outputs or IO's of the design are called as ports
Pins	The primary inputs, outputs, IO's of cells in the design are called as pins
Net	Wires used for the connection between ports of the pins of the different designs are called as net
Clock	The input port or pin used as clock source is called as Clock
Library	The technology specific cells used for targeting for synthesis, linking or for reference are called as library

Once the above variable or parameters are setup for the desired process node library then the synthesis tool can be invoked at the command prompt.

The design objects are described in the above table and are used during synthesis. Every design is the description of the logic circuit to perform some of the logical operations. The design can be single system description or can consist of the multiple sub-systems. The design objects are described in Table 10.1.

## 10.5 Constraining Design Using Synopsys DC

The design is described using VHDL or Verilog languages using the synthesizable constructs. This design need to be used as an input by design compiler. Table 10.2 describes the key commands used by design compiler for various definitions.

### 10.5.1 Reading the Design

It is essential for the ASIC design engineer to understand about the difference between the read command and the analyze, elaborate command? The following are key highlights:

1. The analyze and elaborate command is used in order to pass required parameters while elaborating the design.
2. The read command is used while entering for the pre-compiled designs or netlists in DC.

**Table 10.2** Commands used to read the design

Command [1]	Description
<b>read -format &lt;format_type&gt; &lt;filename&gt;</b>	Used to read the design. For example designer need to read the Verilog module processor.v then the command can be <b>read-format verilog processor.v</b>
<b>analyze -format &lt;format_type&gt; &lt;list of file names&gt;</b>	Used to analyze the design for the syntax errors and translation before building the generic logic. The generic logic is part of the Synopsys generic technology-independent library. The components are named as GTECH. This logic is unmapped representation of the Boolean functions. The command can be used as <b>analyze -format verilog processor.v</b>
<b>elaborate -format &lt;list of module names&gt;</b>	Used to elaborate the design and can be used to specify the different architectures during elaboration for the same analyzed design. The command can be <b>elaborate -library work processor</b>

- Using analyze and elaborate commands the different architectures can be specified during elaboration for the same analyzed design.
- The read command does not allow the use of the different architectures.

### 10.5.2 *Checking of the Design*

After the design has been read using the design compiler, the check\_design is used. Table 10.3 describes the command used to check the errors in the design.

### 10.5.3 *Clock Definitions*

The clock needs to be defined using the command create\_clock and this is used as reference for timing analysis. Table 10.4 describes the clock definition commands.

If designer wish to use the clock for variable duty cycle with rising edge at 1 ns and clock period of 5 ns, then the same command can be modified as

**Table 10.3** Command used to check the design

Command [1]	Description
<b>check_design</b>	Used to check the design problems like shorts, opens, multiple connections and instantiations with the no connections <b>check_design</b>

**Table 10.4** Commands for clock definition

Command	Description
<b>create_clock -name &lt;clock_name&gt; -period &lt;clock_period&gt; &lt;clock_pin_name&gt;</b>	Used to create the clock for the design and used as reference during timing analysis. The clock is always associated with the clock pin of the design. If design does not have clock then it will be treated as virtual clock. The command can be used to generate 200 MHz clock with 50 % duty cycle is <b>create_clock -name clock -period 5 master_clock</b>

**create\_clock -name clock -period 5 -waveform {1,5} -name master\_clock**

If the design does not have the clock pin then the virtual clock is created using following commands:

This command generates virtual clock of frequency 200 MHz with 50 % duty cycle.

**create\_clock -name clock -period 5**

This command generates virtual clock of frequency 200 MHz. with variable duty cycle with rising edge at 1 ns and falling edge at 5 ns.

**create\_clock -name clock -period 5 -waveform {1,5}**

### 10.5.4 Skew Definition

As discussed in the previous section, the skew is difference between arrivals of the clock signal at various pins of the flip-flop. If clock at the source flip-flop is delayed with reference to the destination flip-flop then the skew is called as negative clock skew and useful for the hold. If clock at the destination flip-flop is delayed with reference to the source flip-flop then the skew is called as positive clock skew and useful for the setup. The reason being clock at the destination flip-flop is delayed the data can arrive late.

The design compiler will not be able to synthesize the clock tree; hence to overcome the problem the clock skew is used to model the propagation delay that exists in the clock tree.

**Table 10.5** Commands used for the skew definitions

Command [1]	Description
<b>set_clock_skew -rise_delay &lt;rising_clock_skew&gt; -fall_delay &lt;falling_clock_skew&gt; &lt;clock_name&gt;</b>	This command is used to define the clock skew for the design. This can be described as <b>set_clock_skew -rise_delay 2 -fall_delay 1 master_clock</b>



**Table 10.6** Commands used for the input, output delay definitions

Command [1]	Description
<b>set_input_delay -clock &lt;clock_name&gt; &lt;input_delay&gt; &lt;input_port&gt;</b>	Used to define the input port delay with reference to the clock. To define 1 ns delay with reference to clock, the command can be used as <b>set_input_delay -clock master_clock 1 data_in</b>
<b>set_output_delay -clock &lt;clock_name&gt; &lt;output_delay&gt; &lt;output_port&gt;</b>	Used to define the output port delay with reference to the clock. To define 1 ns delay with reference to clock, the command can be used as <b>set_output_delay -clock master_clock 1 data_out</b>

Table 10.5 describes the commands used by design Compiler while defining clock skew for the design.

### 10.5.5 Defining Input and Output Delay

The input and output delay can be defined by using `set_input_delay` and `set_output_delay` respectively. Table 10.6 describes the command used with the required parameter definition.

### 10.5.6 Defining Minimum (Min) and Maximum (Max) Delay

The input and output delays can be defined as min or max depending on the requirements. Table 10.7 describes the min and max delay definitions.

### 10.5.7 Design Synthesis

Using the command `compile` the design can be synthesized, prior to synthesis the design constraints need to be given to the design. The design can be synthesized using the different efforts levels like low, medium, and high.

Table 10.8 describes the `compile` command.

### 10.5.8 Saving the Design

The design can be saved by using `write` command in various formats using design compiler. The format can be Verilog or Database format (`ddc`).

Table 10.9 describes the command used to save the design.

**Table 10.7** Commands used for min, max IO delay definitions

Command [1]	Description
<b>set_input_delay -clock &lt;clock_name&gt; -max &lt;delay&gt; &lt;input_port&gt;</b>	Used to define the max input port delay with reference to the clock. To define 2 ns delay with reference to clock, the command can be used as <b>set_input_delay -clock master_clock -max 2 data_in</b>
<b>set_input_delay -clock &lt;clock_name&gt; -min &lt;delay&gt; &lt;input_port&gt;</b>	Used to define the min input port delay with reference to the clock. To define 1 ns delay with reference to clock, the command can be used as <b>set_input_delay -clock master_clock -min 1 data_in</b>
<b>set_output_delay -clock &lt;clock_name&gt; -max &lt;delay&gt; &lt;output_port&gt;</b>	Used to define the max output port delay with reference to the clock. To define 2 ns delay with reference to clock, the command can be used as <b>set_output_delay -clock master_clock -max 2 data_out</b>
<b>set_output_delay -clock &lt;clock_name&gt; -min &lt;delay&gt; &lt;output_port&gt;</b>	Used to define the min output port delay with reference to the clock. To define 1 ns delay with reference to clock, the command can be used as <b>set_output_delay -clock master_clock -min 1 data_out</b>

**Table 10.8** Command used for compiling the design

Command [1]	Description
<b>compile -map_effort &lt;map_effort_level&gt;</b>	This command is used to synthesize the design with different effort levels like low, medium, and high. The command for the medium effort level can be <b>compile -map_effort medium</b>

**Table 10.9** Command used to save the gate level netlist

Command [1]	Description
<b>write -format &lt;format_type&gt; -output &lt;file_name&gt;</b>	This command is used to save the output generated by synthesizer in the various formats. For the Verilog format the command can be <b>write -format verilog -output processor_netlist.v</b>

## 10.6 Synthesis Optimization Techniques

Before discussion on the synthesis, timing and reports let us understand the different synthesis techniques used for the optimization. The optimization can be performed at the code level or during synthesis. The fully optimized design is that which has met the area and timing requirements. The optimization at the RTL level can be

achieved by modifying the code to meet the intended functionality. In such type of optimizations care has to be taken that, the optimized code should have the same simulation results before synthesis and after synthesis. But there are few standard techniques used in the real practical scenarios to have better synthesis optimizations and results. A few of such techniques are discussed in this section.

### 10.6.1 Resource Allocation

This is used for the better synthesis results and this optimization technique uses the sharing of hardware resources.

Consider the Verilog procedural block described in the following example

```
always@(*)
begin
if(a_in==1)
y_out = b_in+c_in;
else
y_out = b_in+d_in;
end
```

The above functionality generates two adders: one to perform addition of `a_in` and `b_in` and another to perform addition of `b_in` and `d_in`. It also generates the 2:1 MUX to select one of the outputs of the adder. The synthesis result is shown in Fig. 10.6.

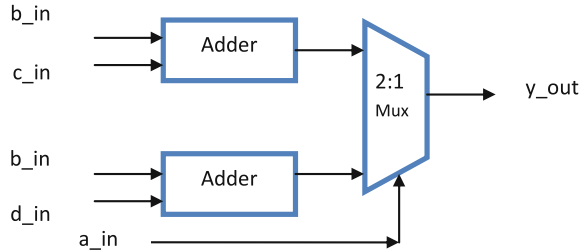
In the above synthesis result, the common input `b_in` is not shared properly. If the above code is modified using only one adder then the synthesis optimization is better due to minimum area. Figure 10.7 shows the synthesis output.

The modified optimized Verilog code functionality is described in the following example.

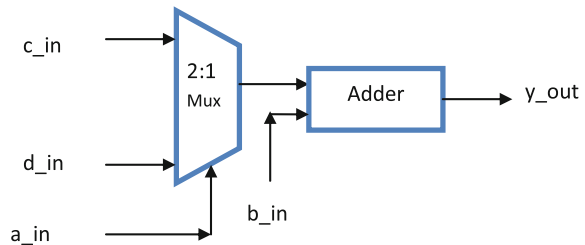
```
always@(*)
begin
if(a_in==1)
y_tmp= c_in;
else
y_tmp= d_in;
end

assign y_out = b_in + y_tmp;
```

**Fig. 10.6** Synthesis result without resource allocation



**Fig. 10.7** Synthesis result with resource allocation



So prior to the sharing of the resources, the area was more but resource sharing technique is effective to reduce the area.

### 10.6.2 Common Factors and Sub-expressions Use for Optimization

In most of the Verilog RTL code, the RTL engineer uses the expressions or sub-expression. Most of the time, the sub-expressions are not reused. If the sub-expression computed values are reused then the synthesizer will be able to perform the better results.

Consider the example shown below. In the following example  $b\_in + c\_in$  is used for the multiple assignments.

```
assign y_tmp= b_in + c_in;
assign z_out = d_in - ( b_in + c_in);
```

Instead of the two assignments the single continuous assignment can give the better logic using minimum resources.

```
assign z_out = d_in -y_tmp;
```

Consider another Verilog RTL code common factor can be reused while writing an efficient Verilog RTL.

```
always@(*)
begin
  if (a_in)
    y_out = b_in & ( c_in + d_in);
  else
    z_out = e_in ^ (c_in +d_in);
end
```

In the above example the common factor is (c\_in + d\_in) and can be reused. The above code can be modified as

```
always@(*)
tmp_add = c_in + d_in;
begin
  if (a_in)
    y_out = b_in & ( tmp_add);
  else
    z_out = e_in ^ (tmp_add);
end
```

These minor modifications in the Verilog RTL can generate more optimized logic.

### 10.6.3 *Moving the Piece of Code*

In most of the Verilog RTL the expressions are used in for or while loops. These expressions values may or may not change during every iteration. Those statements used in for or while loops whose value will not change can be handled by using the changes in the code. The synthesizer during optimization handles such scenario but

there are chances of redundant logic generation. This can be avoided by moving the expression outside of the loop. Consider the following Verilog RTL.

```
//The value of y_tmp in the range of 0 to 9
assign y_tmp = a_in + b_in;
for ( y_tmp =0, y_ymp < 9, y_tmp++)
z_out = y_tmp-9;
```

In the above example it is assumed that `y_tmp` is not assigned with the new value within the loop and the above expression remains constant for every iteration inside the loop. The synthesizer generates the nine subtractors during synthesis and this occupies more area. The above Verilog RTL functionality can be modified to avoid the unnecessary logic.

```
//The value of y_tmp in the range of 0 to 9
assign y_tmp = a_in + b_in;
assign tmp= y_tmp-9
for ( y_tmp =0, y_ymp < 9, y_tmp++)
z_out = tmp;
```

### 10.6.4 Constant Folding

Consider the use of constant in the Verilog RTL code. Instead of writing the code use the direct computed or required value for the `y_out`. The Verilog RTL piece of code is shown in the following example.

```
integer c_in =3;

assign y_out= c_in *3;
```

Instead of using the above unnecessary Verilog RTL, the better way is to use value 9 for `y_out` and this technique is called as constant folding.

### 10.6.5 Dead Zone Elimination

The section of the code which is never executed is called as dead zone code. The dead zone code elimination technique has to be used for the better synthesis results.

The piece of Verilog RTL is shown in the following example

```
integer c_in=3;
integer b_in =2;

always@(*)
if (b_in >c_in)
y_out=1;
else
y_out=0;
end
```

In the above code the condition is always false and hence if statement always generates the false output. The synthesizer during synthesis will perform such kind of optimizations. But if the code is modified then it will reduce the time during synthesis.

### 10.6.6 Use of Parentheses

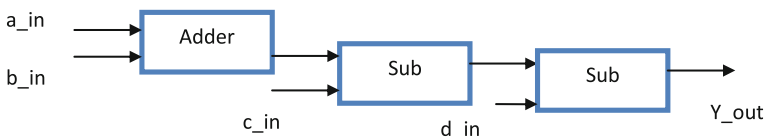
In the most of the Verilog RTL designs if parentheses are used properly then the synthesis results can be better optimized.

For example if the assign statement is used in the Verilog RTL without any parentheses then it generates the logic with more propagation delay (Fig. 10.8).

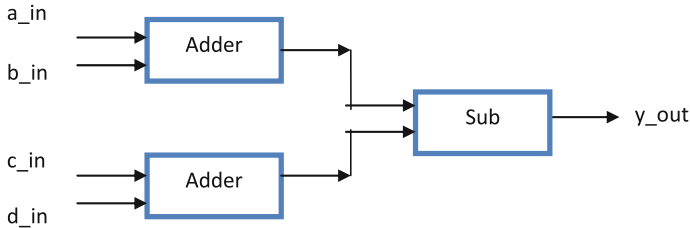
```
assign y_out= a_in + b_in - c_in -d_in;
```

If the above statement is modified as shown below then it gives the clear timing and data path (Fig. 10.9).

```
assign y_out= (a_in+b_in) - (c_in+d_in);
```



**Fig. 10.8** Synthesis result without use of parentheses



**Fig. 10.9** Synthesis result with use of parentheses

### 10.6.7 Partitioning and Structuring the Design

The design needs to be structured and partitioned for the better synthesis outcome. It is the practical reality that the design which is better partitioned generates better synthesis results and even it reduces the synthesis runtime. The following are key guidelines recommended for the design partitioning:

1. Partition the design for the design reuse.
2. For the different functionality use the different module.
3. Use the combinational related logic in the same block.
4. Use the separate block or structure logic and for the random logic.
5. Partition the design at the top level.
6. Do not use the glue logic at the top level.
7. Use the separate module for state machines that is isolating the state machines form the other logic.
8. Limit the logic size to maximum 10 K gates for every block.
9. Avoid use of the multiple clocks in the same block.
10. Isolate the synchronizers for the multiple clock domain designs.

The timing, area reports and synthesis script is discussed in Chap. 12

## 10.7 Summary

Following are the few highlights to summarize this chapter:

1. ASIC stands for Application Specific Integrated Circuit and is used to design the chips for the specific applications.
2. In the full-custom ASIC designs the design does not use the predefined library cells. The design is done from the scratch.
3. In the semi-custom ASIC design the design uses the predefined and tested standard cell library components.
4. RTL synthesis is a process used to convert the Verilog RTL into the gate level netlist.



5. RTL synthesizer uses the Verilog RTL, libraries, and constraints as an input.
6. The physical design flow has the steps like floor planning, power planning, CTS, placement and routing, and back-annotation.
7. The Synopsys DC uses the different optimization techniques while performing the synthesis.
8. The optimization can be achieved by modification in the RTL code or by using the synthesis optimization algorithms.

## Reference

1. [www.synopsys.com](http://www.synopsys.com) Guidelines and practices for successful logic synthesis version 1998, 08 Aug 1998

# Chapter 11

## Static Timing Analysis



STA is non-vectorized approach used in the timing closure. STA is used to find whether all the timing paths are met or not. For RTL design engineer it is essential to have good understanding of different timing paths. This chapter discusses about the STA concepts used in the timing closure.

**Abstract** Static timing analysis (STA) is used for the timing checks for any ASIC designs. The objective of this chapter is to discuss in detail STA concepts used by the timing analyzer. This chapter discusses about the register timing parameters and their use in the frequency calculations. The positive clock skew and negative clock skew are also discussed in detail with the practical scenario. This chapter also focuses on the different timing paths and SDC commands and their use while writing the script. The solutions and techniques to fix the setup and hold violations are also discussed for the better understanding of the engineers. Even the timing exceptions like false and multicycle paths are covered with the practical scenario.

**Keywords** Timing violations • Setup time • Hold time • Clock to q delay • Timing paths • Timing exceptions • Positive clock skew • Negative clock skew • Slack • Data arrival time • Data required time • Logic duplication • Priority encoding • Multiplexed encoding • Register balancing • Pipelining • Asynchronous paths • Hold fixes • Data path • Clock path • Maximum operating frequency

In the previous section we have discussed about the key RTL Verilog concepts and few important synthesis commands in detail. But we have not discussed about the timing parameters for the ASIC design. The timing analysis is very important phase for any ASIC design and it can be performed during various design phases. Timing analysis can be performed before design layout stage and after design layout stage. So it is essential and important to understand key timing considerations for a ASIC design.

Before layout the timing analysis is performed on gate-level netlist of the design with goal of fixing the setup time. Timing analysis tool uses the design constraint file and the vendor libraries to perform the timing analysis for the design. Timing analysis is of two types static and dynamic. Static timing analysis (STA) is performed without using any set of vectors, and dynamic timing analysis is performed using set of vectors for the design. The goal is to fix the setup and hold time violations for the design.

For any sequential element two important design parameters are setup and hold time.

If setup time or hold time is violated, then the design goes into metastable state. So it is essential to find out and fix the timing violations in the design and this process is performed by the timing analysis tool. Popular timing analysis tool is synopsys prime time (Synopsys PT). This section focuses on the key timing considerations their importance during timing closure.

## 11.1 Setup Time

The minimum amount of time required for which the input signal 'd' of flip-flop should maintain stable value either logic '0' or logic '1' before arrival of an active edge of the clock is called as setup time.

The setup time considerations need to be checked for the design when the design is over constrained. There can be many setup violations. The designer can perceive that the violations in the design are due to the too tight constraints in the design.

To meet the setup time it is required that the data should arrive at the input of 'D' flip-flop before certain amount of time before arrival of the active clock edge. For example, if we consider design operated with 200 MHz clock frequency (Clock cycle time period = 5 ns) and has setup time requirement of 1 ns then it is required that data should arrive at least 4 ns with reference to the active edge of clock so that the required setup time of 1 ns can be met.

Consider Fig. 11.1 consisting of combinational logic with the register. If setup time is  $t_{su}$  then the data arrival time which is dependent of  $t_{comb}$  should be such that it should meet the setup time.

So the required time to travel data at 'D' input is  $T_{clk} - t_{su}$ . The data arrival time is  $t_{comb}$ , that is, delay of combinational logic.

Figure 11.2 shows the valid setup time region with the condition to meet the desired setup time. Consider the positive edge of the clock the data arrives at the D input of flip-flop prior to setup time window. So there is no setup violation in the design.

Data arrival time is the amount of time required to arrive the data at the data input of the D flip-flop. It is given by

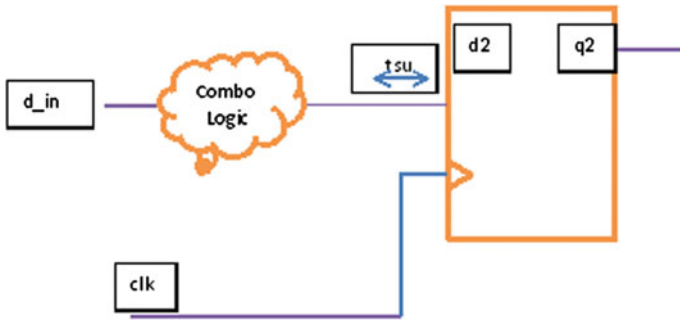


Fig. 11.1 Input-to-register path



Fig. 11.2 Valid setup time region timing sequence

Data arrival time = Propagation delay of flip-flop + combinational delay

Data required time = time duration of clock cycle – setup time

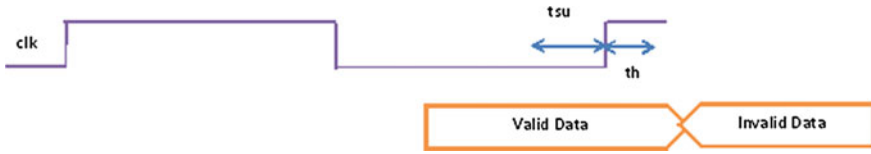
The difference in between data required time and data arrival time is called as slack and to meet setup time the slack should be positive.

## 11.2 Hold Time

The amount of time for which the input signal ‘d’ of flip-flop should maintain the stable value either logic ‘0’ or logic ‘1’ after arrival of an active edge of the clock is called as hold time.

Hold time is important timing parameter consideration in the design. For most of the designs constrained at high frequency it is critical to meet the hold parameter. During the STA at ASIC layout stages most of the hold violations are reported and fixed. The hold violations in the design are due to the fact that data is arriving slowly as compared to the required time.

For example, consider the scenario in Fig. 11.3. If the design is constrained at 200 MHz operating frequency that is clock cycle time is 5 ns. If hold time requirement is 1 ns and data arrived at ‘D’ input of flop changes during the 1 ns window after arrival of active clock edge, then there exist hold violations in the design.



**Fig. 11.3** Valid hold time region timing sequence

As shown in Fig. 11.3 the valid constant data is present on the D input of the register during setup and hold time durations. Both setup and hold times are met for the said design; hence, there is no timing violation in the design.

Data arrival time = Propagation delay of flip-flop + combinational delay should be greater than hold time of flip-flop.

If propagation delay of flip-flop is 3 ns and combinational delay is 1 ns for the design, then data will never change during the 1 ns window, so there is no chance of hold violation in the design.

But consider the scenario in the design that flip-flop propagation delay is 0.8 ns but hold time is 1 ns and if there is no combinational logic in the data path; then the hold violation occurs in the design.

So with reference to our discussion, it is important to note that the data should be stable at the D input of register during setup and hold time windows.

## 11.3 Clock to Q Delay

The amount of time required for the flip-flop to generate valid output either logic '0' or logic '1' after arrival of an active clock edge is called as propagation delay of flip-flop. Propagation delay of flip-flop is also called as clock to 'q' delay.

Consider  $t_{su}$  is the setup time of flip-flop,  $t_h$  is the hold time of flip-flop, and  $t_{pff}$  is the propagation delay of flip-flop. Figure 11.4 describes the various timing parameters for the register.

### 11.3.1 Frequency Calculations

As shown in Fig. 11.4 the timing parameters of flip-flop (reg1) are given as  $t_{pff1}$  and  $t_{su1}$ , and the timing parameters for the register 2 are given as  $t_{pff2}$  and  $t_{su2}$ . The combinational logic design delay in the data path is given as  $t_{comb}$ .

These timing parameters are used to find out maximum operating frequency for the design.

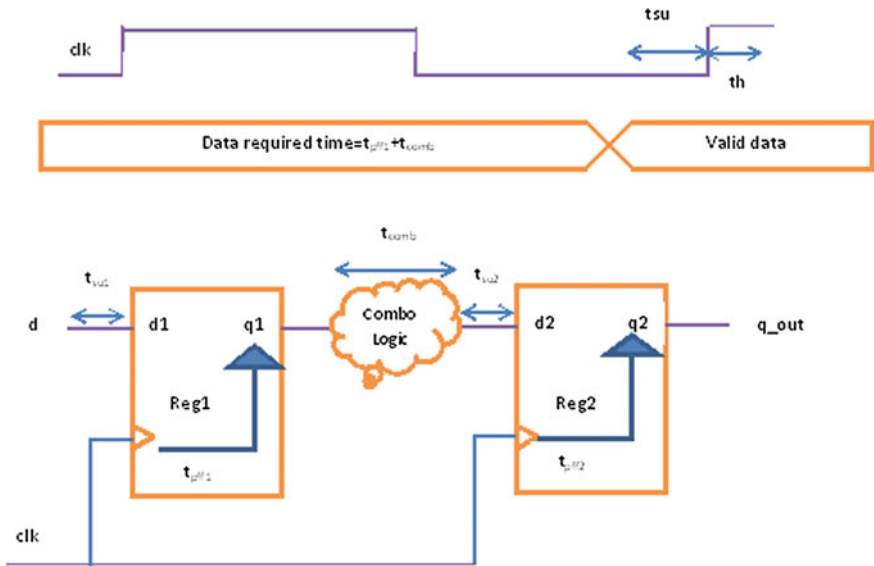


Fig. 11.4 Register-to-register path

To find the maximum operating frequency for the design, find out the data required time and data arrival time. The data required time is the addition of all the delays in the register-to-register path.

Therefore, the data required time is given by  $t_{pff1} + t_{comb}$ .

The data arrival time is given by  $T_{clk} - t_{su2}$ , where  $T_{clk}$  is the one clock cycle time and  $t_{su2}$  is the setup time of second flip-flop.

So the maximum frequency is calculated by equating the data required time and data arrival time.

$$t_{pff1} + t_{comb} = T_{clk} - t_{su2}$$

$$T_{clk} = t_{pff1} + t_{comb} + t_{su2}$$

$$F_{max} = 1 / (t_{pff1} + t_{comb} + t_{su2})$$

Consider both registers have same timing parameter values, that is,  $t_{pff1} = t_{pff2} = 2 \text{ ns}$ ,  $t_{su1} = t_{su2} = 1 \text{ ns}$ , and  $t_{comb} = 2 \text{ ns}$ . Then the maximum operating frequency is

$$F_{max} = 1 / (2 + 2 + 1) \text{ ns} = 200 \text{ MHz}$$

### 11.4 Skew in Design

Consider the example shown in Fig. 11.5. In this example the register 1 is triggered early and register 2 is triggered late. Register 1 is called as launch flip-flop and register 2 is called as capture flip-flop. As the launch flip-flop is triggered first and capture flip-flop is triggered last, there is skew in the clock pulse and it is called as positive clock skew.

In the above example, clock and data travels in the same direction and due to buffer delay the clk1 is delayed by delay of buffer as compared to clk input of register Reg1.

To find the maximum operating frequency for the above design, find out the data required time and data arrival time. The data required time is the addition of all the delays in the register-to-register path.

Therefore, the data required time is given by  $t_{pff1} + t_{comb}$ .

The data arrival time is given by  $T_{clk} - t_{su2} + t_{buf}$ , where  $T_{clk}$  is the one clock cycle time and  $t_{su2}$  is the setup time of second flip-flop where  $t_{buf}$  is the buffer delay of the buffer in the clock path.

So the maximum frequency is calculated by equating the data required time and data arrival time.

$$t_{pff1} + t_{comb} = T_{clk} - t_{su2} + t_{buf}$$

$$T_{clk} = t_{pff1} + t_{comb} + t_{su2} - t_{buf}$$

$$F_{max} = 1 / (t_{pff1} + t_{comb} + t_{su2} - t_{buf})$$

Consider both registers have same timing parameter values, that is,  $t_{pff1} = t_{pff2} = 2$  ns,  $t_{su1} = t_{su2} = 1$  ns,  $t_{buf} = 1$  ns, and  $t_{comb} = 2$  ns. Then the maximum operating frequency is

$$F_{max} = 1 / (2 + 2 + 1 - 1) \text{ ns} = 250 \text{ MHz}$$

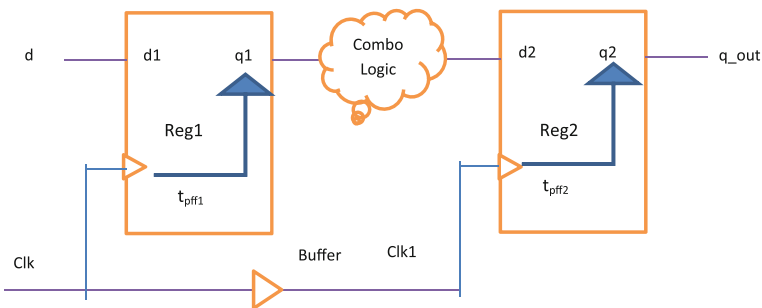


Fig. 11.5 Positive clock skew in the design

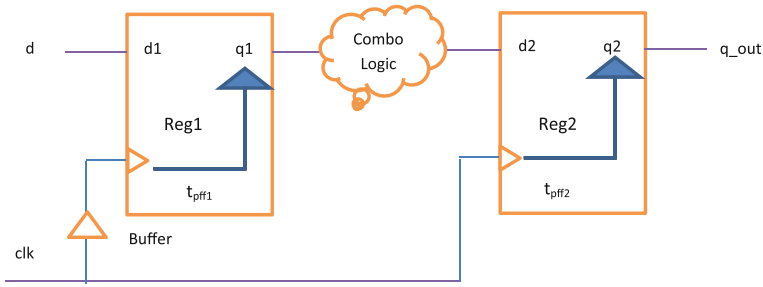


Fig. 11.6 Negative clock skew in the design

So from the above discussion it is clear that positive skew is good to improve the performance of design. In the above example due to the buffer delay of 1 ns the clock at register 2 is delayed by 1 ns time as compared to the clk at register 1. So the time of 1 ns delayed clock can be compensated by setup time and hence increases frequency by 50 MHz.

Let us consider another example shown in Fig. 11.6. In this example source flip-flop is triggered last and destination flip-flop is triggered first. In the other way one can perceive that the clock and data are traveling in the opposite direction.

To find the maximum operating frequency for the above design find out the data required time and data arrival time. The data required time is the addition of all the delays in the register-to-register path.

Therefore, the data required time is given by  $t_{pff1} + t_{comb} + t_{buf}$ .

The data arrival time is given by  $T_{clk} - t_{su2}$ , where  $T_{clk}$  is the one clock cycle time and  $t_{su2}$  is the setup time of second flip-flop where  $t_{buf}$  is the buffer delay of the buffer in the clock path.

So the maximum frequency is calculated by equating the data required time and data arrival time.

$$t_{pff1} + t_{comb} + t_{buf} = T_{clk} - t_{su2}$$

$$T_{clk} = t_{pff1} + t_{comb} + t_{su2} + t_{buf}$$

$$F_{max} = 1 / (t_{pff1} + t_{comb} + t_{su2} + t_{buf})$$

Consider both registers have same timing parameter values, that is,  $t_{pff1} = t_{pff2} = 2$  ns,  $t_{su1} = t_{su2} = 1$  ns,  $t_{buf} = 1$  ns, and  $t_{comb} = 2$  ns. Then the maximum operating frequency is

$$F_{max} = 1 / (2 + 2 + 1 + 1) \text{ ns} = 166.66 \text{ MHz}$$

So from the above discussion it is clear that negative clock skew degrades the performance of design. In the above example due to the buffer delay of 1 ns the



clock at register 1 is delayed by 1 ns time as compared to the clk at register 2. So the time of 1 ns delayed clock is added in the data path with the register delay and hence reduces the clock operating frequency for the design.

## 11.5 Timing Paths in Design

As discussed in the above section the STA is non-vectorized approach to check the timing performance of the ASIC design by checking the timing violations in all possible timing paths.

Timing paths in design start at 'Start point,' clock port of the register or input port of the design is treated as start point. Timing path terminates or ends at the 'End point,' and data input of register or an output port is treated as end point.

For any RTL design there can be four timing paths and they are named as

- Input-to-register path (Input-to-reg path)
- Output-to-register path (output-to-reg path)
- Register-to-register path (Reg-to-reg path)
- Input-to-output path (combinational path)

So timing analyzer checks for the worst possible delays through each of the logic elements in the timing paths but ignores the logical operations. As timing analyzer ignores the logic operations it is non-vectorized approach and more faster as compared to the simulator. But reader needs to understand that the timing analysis is used to check for the timing correctness of the design but not used to check for the logical functional correctness for the design.

This section discusses about the different timing paths in the design.

### 11.5.1 *Input-to-Register Path*

Input-to-register path has start point input port 'q1' and end point data input 'd2' of the register element. This path is also called as input register path group. Figure 11.7 shows the input port 'q1' and combinational logic (Combo logic) and the path from 'q1' to 'd2' through combo logic is treated as input-to-register path.

### 11.5.2 *Register-to-Output Path*

Register-to-output path has start point clock input port 'clk' and end point data output 'q\_out' of the register element. This path is also called as output register path group. Figure 11.8 shows the start point port 'clk' and data 'd' travels through the register through combinational logic and hence named as register-to-output path.

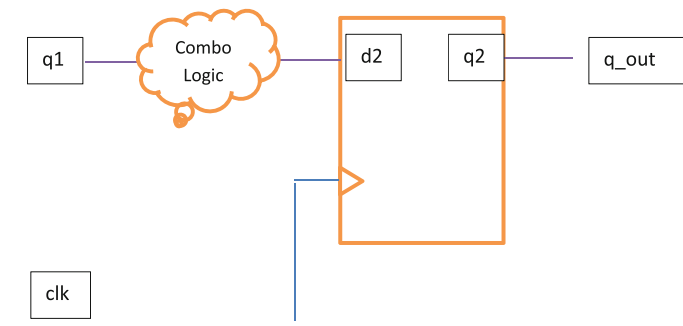
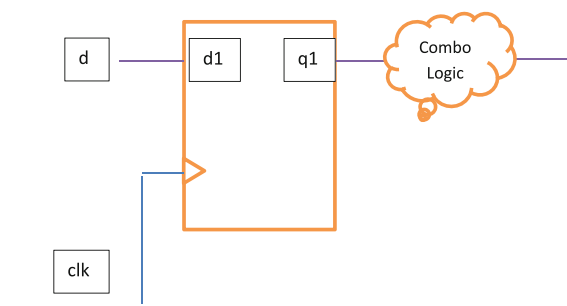


Fig. 11.7 Input-to-register path

Fig. 11.8 Register-to-output path



### 11.5.3 Register-to-Register Path

Register-to-register path has start point clock input port 'clk' and first register acts as an launch register end point data input 'd2' of the second register element. This path is also called as clock path group. Figure 11.9 shows the clock port 'clk' and

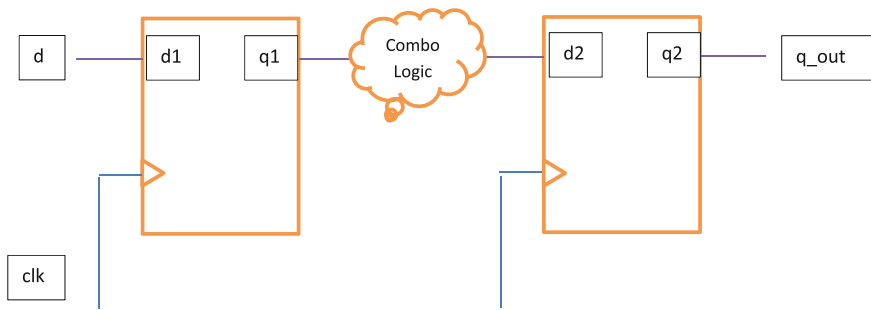
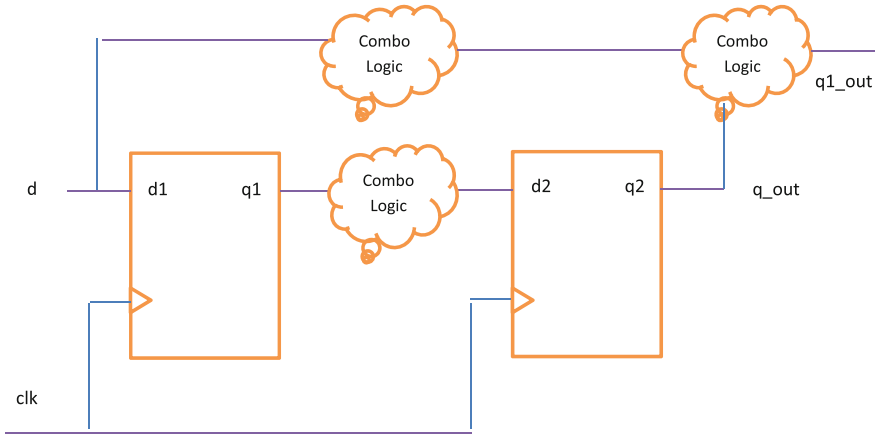


Fig. 11.9 Register-to-register path



**Fig. 11.10** Input-to-output combinational path

launched data by register 1 passes through the combinational logic (Combo logic) before arriving at the data input ‘d2’ of the second register. This path decides the maximum operating frequency for the design.

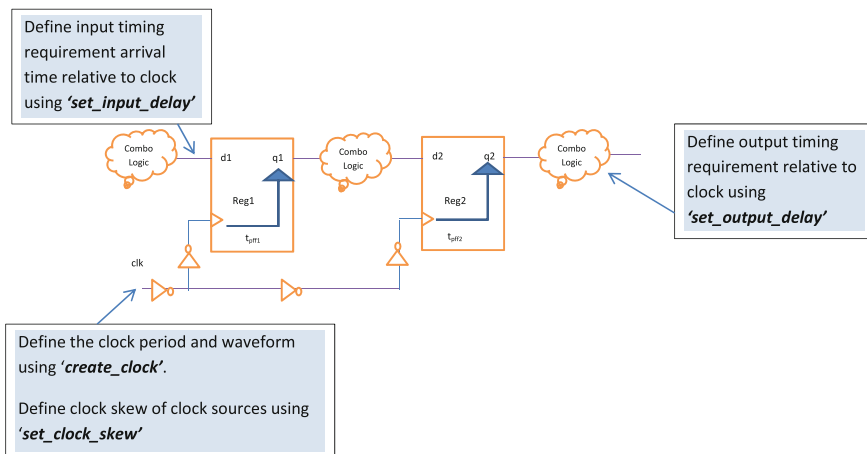
#### 11.5.4 Input-to-Output Path

Input-to-output path has start point input port ‘d’ and end point data output ‘q1\_out.’ This path is also called as combinational path group. Figure 11.10 shows the input port ‘d’ and the data passes through the combinational logic (Combo logic) to generate an output ‘q1\_out.’

### 11.6 Timing Goals for the Design

In the practical scenario the design timing goals are described using the clock definitions for the design and by defining IO timing relative to the clock. The reason for all this definition in the synchronous designs as data arrives from the clocked device and the data goes to the clocked device.

The following template shown in Fig. 11.11 describes the definition required to specify the timing goals for the design.



**Fig. 11.11** Timing goals for synchronous design

Use the SDC commands to define the clock, input delays, output delays, and clock skew.

The SDC [1] commands to specify the timing goals are listed in Fig. 11.12.

## 11.7 Min-Max Analysis for ASIC Design

So from the above discussion it is clear that the setup time is due to faster clock arrival and slow data arrival. To overcome the setup violations the data should arrive fast, launch clock should arrive fast, and capture clock should arrive slowly.

Hold time violation is due to that data arrival is fast, capture is slow, and data arrival is fast. The hold time can be fixed using the slow data arrival, launch is slow, and capture is fast.

In the practical scenario, the min, max corner analysis can be performed using minimum value of timing parameters and using maximum value of timing parameters. During setup time analysis consider the maximum data path delays and minimum delays in the clock path. During hold analysis consider minimum delays in the data path and maximum delays in the clock path.

The following example shown in Fig. 11.13 is used to describe the minimum, maximum analysis for the design.

In this example the minimum delays are considered in the clock path and maximum delays are considered in the data path. Consider the timing parameters of register 1 and register 2.

Consider the first register delay as (1.35, 1.5) ns and the second register delay as (1.65, 1.75) ns and the combinational path delay as 2 ns. Inverter propagation delay is (0.75, 0.8) and setup time of both the register is (0.6, 0.65).

**Fig. 11.12** SDC commands to specify timing goals

- Define the clock for 200MHz operating frequency and having 50% duty cycle by using

```
create_clock -period 5.00 -name clk [ get_ports {clk} ]
```

The above SDC command generates the clock of 200MHz with the 50% duty cycle that is on time is equal to off time.

- Define the clock latency. For example if the clock latency is of 1ns then use the command 'set\_clock\_latency'

```
set_clock_latency -source 1.00 [ get_clocks clk]
```

- Timing analyzer uses the longest or shortest path during timing analysis. The longest delay path is specified by -late and shortest delay path is specified by the -early path.
- The setup analysis the timing analyzer uses the late clock latency for the data arrival path and early clock latency for the clock arrival path. The clock latency for setup is defined with reference to rising (-rise) or falling (-fall) clock transitions.
- For the hold analysis the timing analyzer uses the early clock latency for the data arrival time and late clock latency for the clock arrival time.
- The definitions for the clock latency can be specified by the following SDC

```
set_clock_latency -source -early -rise -0.5 [ get_clocks clk]
```

```
set_clock_latency -source -early -fall -0.45 [ get_clocks clk]
```

- Specify the separate clock uncertainty for the setup (-setup) and for the hold (-hold)

```
set_clock_uncertainty -setup 1.0 [ get_clocks clk]
```

```
set_clock_uncertainty -hold 0.5 [ get_clocks clk]
```

- Specify the minimum and maximum input delays for the design using 'set\_input\_delay'

```
set_input_delay -clock clk -max 2.0 find (port d1)
```

```
set_input_delay -clock clk -min 1.4 find (port d1)
```

- Specify the minimum and maximum output delays for the design using 'set\_output\_delay'

```
set_output_delay -clock clk -max 1.8 find (port q_out)
```

```
set_output_delay -clock clk -min 1.2 find (port q_out)
```

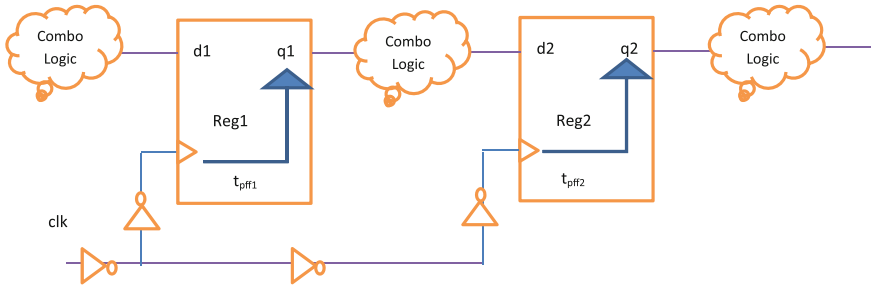


Fig. 11.13 Min-max delay analysis

Skew in the design is due to the inverters in the clock path. This skew is calculated as follows. Using minimum delay analysis the skew in the design is  $1.2 - 0.6 = 0.6$  ns. This skew is due to additional delay of inverter for the capture flop.

Data arrival time is equal to  $T_{pff1} + T_{combo} = 1.5 + 2 = 3.5$  ns

Data required time is equal to  $t_{clk} + t_{skew} - t_{su} = t_{clk} + 0.6 - 0.6$ . Then the maximum operating frequency due to minimum time period of design is

$$T_{pff1} + T_{combo} = t_{clk} + t_{skew} - t_{su}$$

$$t_{clk} = T_{pff1} + T_{combo} - t_{skew} + t_{su} = 1.5 + 2 - 0.6 + 0.6 = 3.5 \text{ ns}$$

$$F_{max} = 1 / (3.5 \text{ ns}) = 285.71 \text{ MHz}$$

The above calculation is for the setup analysis that is maximum delay in the data path and minimum delay in the clock path.

## 11.8 Fixing Design Violations

As discussed in the Sects. 11.1–11.7 the set up time and hold time should be met for all the registers in the design. If setup and hold time is not met then it results into timing violation. Following are few techniques used to fix the design violations.

### 11.8.1 Changes at the Architecture Level

To fix the design violations the last option is to make the required and necessary changes at the architecture level of design. But the architecture level changes are not recommended for the design as it can have significant impact on the design and implementation cycle. But after incorporating changes at the microarchitecture of the design or during optimization if the timing constraints are not met, sometime it

is essential to incorporate the changes at the architecture level. The designer needs to suggest to chief architect about the required changes in the architecture. The chief architect needs to take care of the design functionality as the changes in the architecture can affect the design functionality. It is essential to make the desirable changes by keeping the same design functionality.

### ***11.8.2 Changes at Microarchitecture Level***

If the design optimization fails to meet the required timing, then it is essential to make the necessary and required changes at the microarchitecture level. The microarchitecture document is the golden reference document for the RTL design and due to that the designer has insight about it. The greater detail understanding of the microarchitecture always plays crucial and significant role during the RTL design stage.

### ***11.8.3 Optimization During Synthesis***

Synthesizers used at the RTL level are more efficient due to the in-built synthesis efficient and optimization algorithms. They are driven by the coding and design styles adopted at the synthesis level. If design does not meet the timing then optimization algorithms need to be used. To meet the desire timing goals the designer can use the optimization concepts like pipelining, register duplications, register balancing, etc. Consider the scenario, if the design needs to be optimized for 100 timing violations, and among them 20–30 timing violations are not possible to fix using synthesis optimizations, then the better approach can make the necessary and required changes in the RTL code to fix these violations.

The readers need to ask themselves that why it is challenging to fix the timing violations in the design? As the design complexity increases from block level to chip level due to multiple hierarchies in the design the propagation delay between registers increases and it has overall significant impact on register-to-register timing. It may be possible that the multiple timing paths can be violated due to non-meeting of the setup and hold time parameters.

It is general observation that at the block level design meets the timing goals, that is, the design does not have any timing violations at the block level. But at the top level design due to integration of multiple blocks there exist possibilities of several timing violations. At the top level these violations can be fixed by minimizing the logic levels between the registers. If data required time is greater than the data arrival time, then it is treated as clean register-to-register path due to positive slack. This indicates that there is no setup violation in the design at top level.

## 11.9 Fixing Setup Violations in the Design

Following are few techniques used to fix the setup time violations:

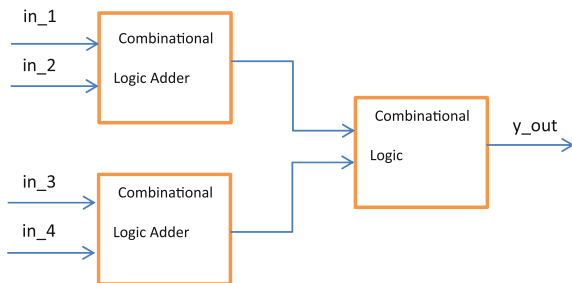
1. Logic duplication
2. Encoding methods
3. Late arrival signal fixes
4. Register balancing

### 11.9.1 Logic Duplication

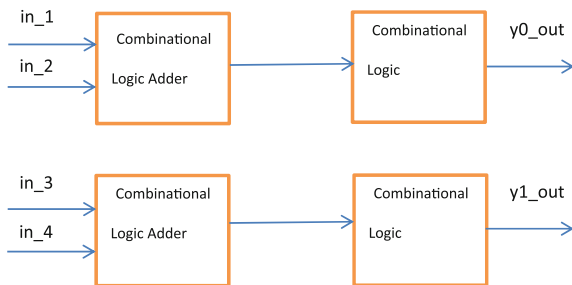
This technique increases the effective area but generates two independent paths during synthesis. This technique is effective to fix setup time violation. For example, consider Fig. 11.14. Consider inputs as in\_1, in\_2, in\_3, and in\_4 are registered inputs and the combinational logic is in the register-to-register path. If every adder has propagation delay of 3 ns, then overall combinational path delay is 6 ns. But due to logic duplication the two independent paths can be optimized to improve the timing.

As shown in Fig. 11.15 the two independent paths have been created using logic duplication technique and hence the optimization for these two independent paths is possible by retaining same functionality. Logic duplication technique increases the area.

**Fig. 11.14** ASIC logic without duplication



**Fig. 11.15** ASIC logic with logic duplication





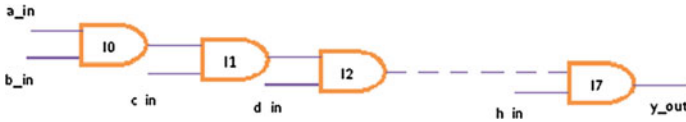


Fig. 11.16 Priority encoding logic

### 11.9.2 Encoding Methods

The popular used encoding techniques are priority encoding and multiplexed encoding. Consider the continuous assignment statement.

```
assign y_out=a_in && b_in && c_in && d_in && e_in && f_in
&& g_in && h_in;
```

The above statement generates the priority logic, where h\_in has highest priority over any other input signal. It generates equivalent logic as shown in Fig. 11.16.

In the priority encoding method the overall delay is of 7tpd, and if tpd is equal to 1 ns then the overall propagation delay is of 7 ns.

To improve the design performance it is essential to reduce the propagation delay of combinational logic and hence multiplexed encoding technique can be efficient as compared to the priority encoding technique. Figure 11.17 shows the multiplex encodings using the continuous assignment.

```
assign y_out= ( (a_in && b_in) && (c_in && d_in) ) && ( (e_in
&& f_in) && (g_in && h_in);
```

As shown in Fig. 11.17 the number of levels has been reduced from seven to three and hence the overall propagation delay for the multiplexed encoding is only 3tpd. If the tpd is 1 ns then overall propagation delay for the multiplexed encoding

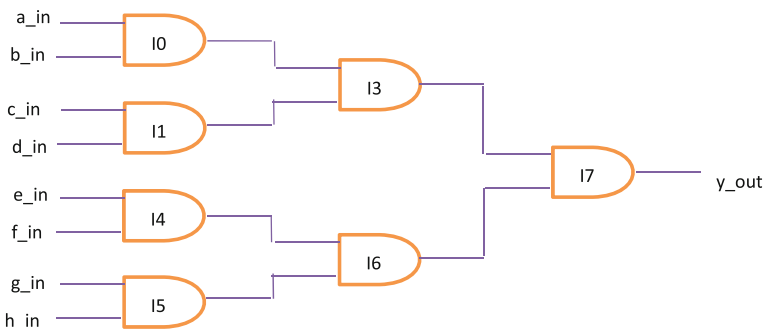
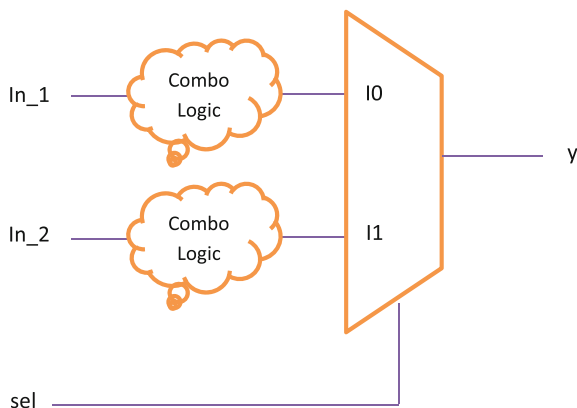


Fig. 11.17 Sequential or multiplexed encoding logic

**Fig. 11.18** Logic with late arrival of signals



is only three-stage delay, that is, 3 ns. So this technique has improved the performance as compared to the priority encoding technique.

### 11.9.3 Late Arrival Signals

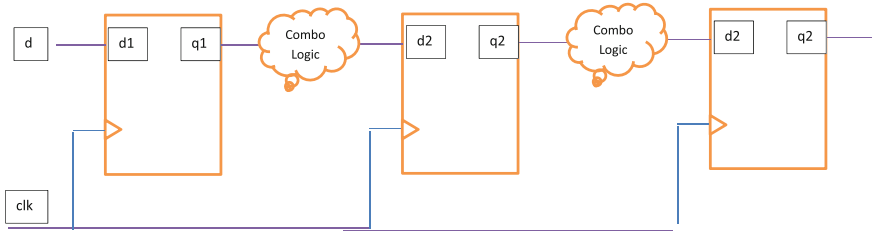
For any design if control signals are late arriving then it has significant impact on the design timing. Due to late arrival of the control signal setup time may be violated.

In the example shown in Fig. 11.18, in\_1 and in\_2 are multiplexer inputs and arrive quickly but sel\_in is select line of multiplexer and arrives very late. The select input sel\_in is late arrival signal. This signal has significant impact on the setup time of design.

To improve the timing and to avoid the setup time violation the combinational logic can be pushed ahead and the multiplexer logic can be pushed later. The combinational logic can be duplicated at the input of multiplexers. This technique increases area but improves the overall design performance by compensating the time required for the combinational logic and late arrival signal. Another important point to understand is this technique allows the logic partitioning efficiently into two groups for further improvement in the timing.

### 11.9.4 Register Balancing

To fix the setup time and to improve the design performance register balancing is one of the powerful techniques. Consider the operating frequency for the design as 200 MHz, that is, clock time period is of 5 ns. If register-to-register path has high



**Fig. 11.19** Register balancing example

combinational delay due to which the data arrival time is greater than the data required time, the slack is negative and it violates the setup time for the design.

Consider the example shown in Fig. 11.19, register 1 to register 2 path has combinational logic and has a delay of 3 ns. If we consider the setup time of register as 1 ns and propagation delay of flip-flop as 2 ns and hold time as 0.5 ns, then the data arrival time for register 1 to register 2 path is 5 ns and data required time is  $T_{\text{clk}} - 1$  ns. So the clock time period  $T_{\text{clk}} = 6$  ns. This violates the setup time of design for the given design constraints of 5 ns.

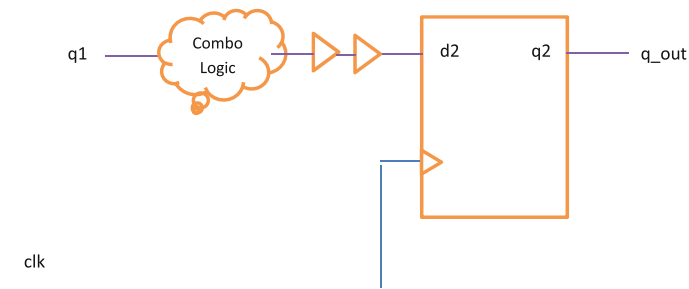
For register 2 to register 3 path the combinational delay specified is 1 ns and if we consider same timing parameters for the register then the data required time is  $T_{\text{clk}} - 1$  ns and data arrival time is 3 ns. This meets the timing constraints for the design. For register 2 to register 3 paths the data is arrived at the D input of register 3 at 3 ns and waiting for the clock which is arriving after 2 ns. So there is additional time margin of 1 ns and this can be used to improve the design performance and this technique is called as balancing the timing between two register-to-register paths.

This can be achieved by splitting the combinational logic between the register 1 and register 2 into two paths and pushing the combinational logic with delay of 1 ns to the register 2 to register 3 path.

This will give the clean timing for all register-to-register paths as the data arrival time for both the paths will be 4 ns. This meets the design constraints and the operating frequency for the design meets the target of 200 MHz.

## 11.10 Hold Violation Fix

Hold time violation occurs in the design if the data at the D input of register changes very fast. For example, consider the design as shown in Fig. 11.1, if combinational logic delay is less and due to that if data at D input of register changes very fast, then there exist hold violation for the design. During the hold time window if the data changes then there is hold violation.



**Fig. 11.20** Hold violation fix

To fix the hold violation for the design it is recommended to insert the buffers in the data path but care need to be taken that this should not violate the setup time requirements for the design. Inserting buffers into the data path increases the time required to change data at the D input of register and thus hold violation can be fixed. The logic after inserting the buffers in the data path is shown in Fig. 11.20.

## 11.11 Timing Exceptions in the Design

There are two main timing exceptions and are named as false paths and multicycle paths. These timing exceptions need to be reported to timing analyzer using SDC commands.

### 11.11.1 Asynchronous and False Paths

If the changes on any one of the signals or ports do not affect on the output of design then the path needs to be reported as false path. False path is basically timing exception and needs to be notified to the synthesis tool. For example, consider the following expression:

```
assign y_out = (a_in + b_in) (c_in + d_in)
```

In this example if the d\_in is set to zero due to some reason, then the logical output depends on only a\_in, b\_in, and c\_in inputs, and the path from d\_in to y\_out will be considered as false path.

*Asynchronous path* Asynchronous path needs to be notified to the synthesis tool and these path violations need to be treated as false violations and need to be ignored.

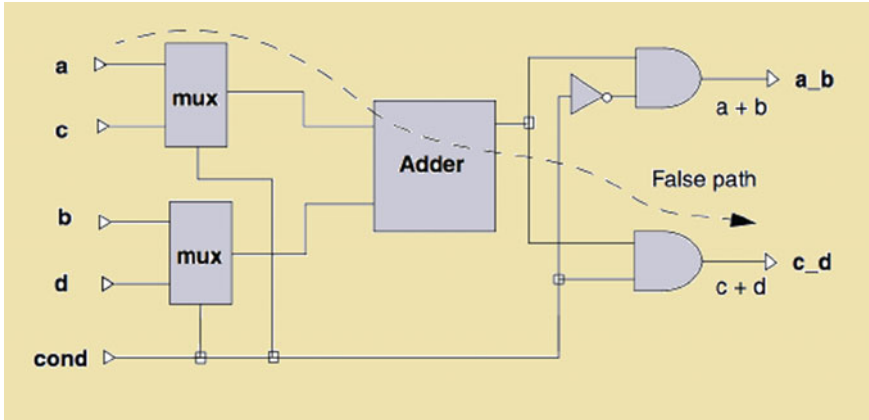


Fig. 11.21 False path example [1]

Figure 11.21 describes the false path and this needs to be reported to the timing analyzer. The SDC command discussed in Chap. 10 can be used to specify the false path.

```
set_false_path -from [ get_ports {a b} ] -to [ get_ports c_d ]
```

*The above SDC command indicates the changes on input ports 'a', 'b' will not affect on the output 'c\_d' and need to be treated as false path*

```
set_false_path -from [ get_ports {c d} ] -to [ get_ports a_b ]
```

*The above SDC command indicates the changes on input ports 'c', 'd' will not affect on the output 'a\_b' and need to be treated as false path*

### 11.11.2 Multicycle Paths

If any path in the design has delay of more than once clock cycle then the path is treated as multicycle path. Consider the following design scenario where register (FF4) to register (FF5) delay is of 40 ns and clock period is of 5 ns. To update the d input of register with new value the number of clock pulses required is equal to 8. This needs to be informed to the tool so that setup and hold windows can be pushed according to the requirement. The multicycle path is a timing exception.

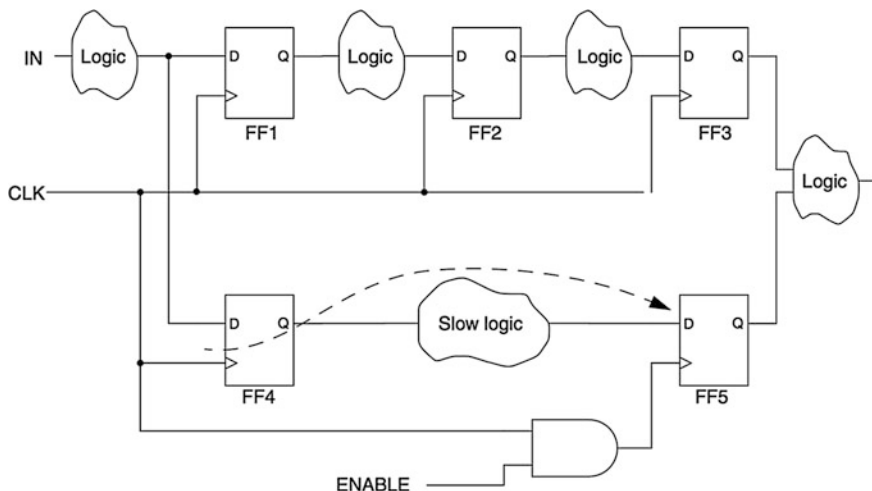


Fig. 11.22 Multicycle path example [1]

The SDC command discussed in Chap. 10 can be used to specify the multicycle path (Fig. 11.22).

```

set_multicycle_path -setup 2 -from [ get_cells FF4 ] \
  -to [ get_cells FF5 ]
The above SDC command indicates that the path specified is
multicycle path and the setup is pushed by 2 clock cycles
set_multicycle_path -hold 1 -from [ get_cells FF4 ] \
  -to [ get_cells FF5 ]
The above SDC command indicates that the path specified is
multicycle path and the setup is pushed by 1 clock cycles

```

## 11.12 Pipelining and Performance Improvement

The design performance for the design can be improved by adding the multiple stage pipelining in the ASIC design. The overall latency to get an output data is dependent upon the number of pipelined stages. Pipelining will increase the area as register utilization for multiple bits increases.

Due to use of pipelining the overall performance of the design also improves. Readers are requested to refer Chap. 6 for better understanding of the pipelining.

### 11.13 Summary

The following are key important points need to be remembered by the ASIC design engineers:

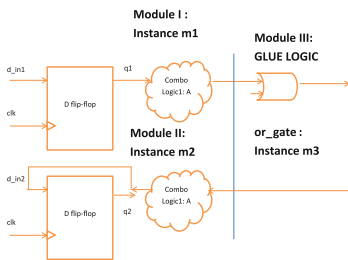
1. STA is non-vectored approach and faster as compared to simulator.
2. Flip-flop timing parameters are setup, hold, and clock to q delay.
3. If setup or hold time is violated then design goes into the metastable state.
4. There are four timing paths in the design and register-to-register path decides the maximum operating frequency for the design.
5. The setup analysis, the timing analyzer, uses the late clock latency for the data arrival path and early clock latency for the clock arrival path. The clock latency for setup is defined with reference to rising (-rise) or falling (-fall) clock transitions.
6. For the hold analysis the timing analyzer uses the early clock latency for the data arrival time and late clock latency for the clock arrival time.
7. The multicycle paths and false paths are the timing exceptions.

### Reference

1. Synopsys Timing Constraints and Optimization User Guide, version D-2010.03

# Chapter 12

## Constraining ASIC Design



Synopsys Design Compiler is industry leading design synthesis tool. Most the leading ASIC design companies uses the Synopsys DC for the synthesis and Synopsys PT for the timing analysis. This chapter focuses on the design constraints for the area, speed and power.

**Abstract** This chapter discusses about the constraining design using Synopsys DC compiler. Every ASIC design needs to meet the constraints. The constraints are classified as optimization, design rule, and environmental constraints. This chapter covers the area minimization techniques, design optimization techniques using the meaningful practical design scenarios. Even this chapter describes about the key important commands used to boost the design performance. This chapter even discusses about the commands used for the FSM extractions. The sample scripts are given in the chapter and can be used for the design optimization and the report generations.

**Keywords** Area · Speed · Power · Register balancing · Optimization · Synthesis · DRC · FSM · Flattening · Grouping · Extraction · Characterize · Max area · Compile · Timing report · Area report · Constraints · Environmental conditions · Slack · Data arrival time · Data required time · Power · Leakage · Dynamic · Capacitance · Load · Fanout



## 12.1 Introduction to Design Constraints

Modern ASIC SOCs are very complex in the nature and consists of more than 100 K gates. Design complexity has grown exponentially in the past decade due to the demand of the sophisticated and intelligent devices. In such scenario there is additional overhead and cost during the design synthesis and timing closure. As discussed in Chap. 10, the ASIC design passes through various phases which include architecture exploration, microarchitecture design, design entry using HDL, simulation, and synthesis. The Synopsys DC is leading EDA tool used to synthesize design and Synopsys PT is used for the timing closure.

As a ASIC design engineer, it is required to have exposure about the design synthesis and timing analysis. These concepts are covered in the Chaps. 10 and 11, respectively. The real understanding of the design constraints and the commands used to constrain the design for the area, speed, and power is very much required to design a chip. This chapter is focused on the design constraints using Synopsys DC.

The design constraints are classified as design rule constraints and optimization constraints. The classification is shown in Fig. 12.1.

Synthesis flow is discussed in Chap. 10 with the key sdc commands. For easy reference the synthesis flow is shown in Fig. 12.2. These are also treated as the steps while carrying out synthesis for any design. The compilation strategy can be chosen as top-down or bottom-up. The commands used at each step are discussed subsequently.

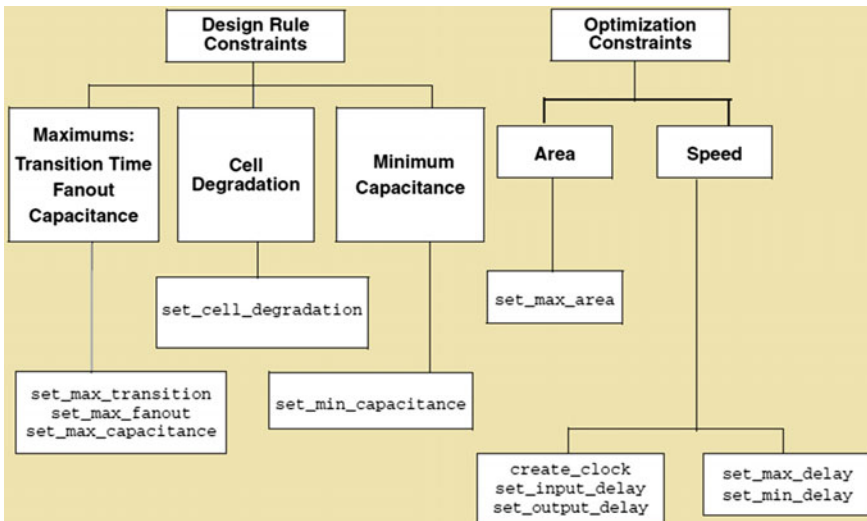
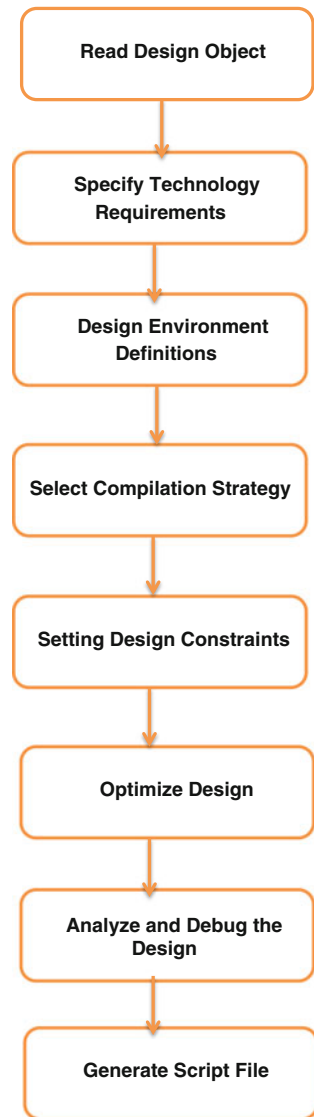


Fig. 12.1 Constraint classification [1]

**Fig. 12.2** Flow for synthesis and optimization



1. Read Design Object: Design object is Verilog RTL code which is simulated for the functional correctness. The commands used at this step are

***analyze, elaborate, read***

2. Specify Technology Requirements: In these steps the design rules and libraries required need to be specified. The commands used in this step are

***Library Objects***

*link\_library*

*target\_library*

*symbol\_library*

***Design Rules***

*set\_max\_transition*

*set\_min\_transition*

*set\_max\_fanout*

*set\_min\_fanout*

*set\_max\_capacitance*

*set\_min\_capacitance*

3. Design Environment Definitions: The design environment includes the process, temperature, voltage conditions, drive strength and effect of load driving the design. The commands used are

***set\_operating\_conditions***

*set\_wire\_load*

*set\_drive*

*set\_driving\_cell*

*set\_load*

*set\_fanout\_load*

4. Select Compilation Strategy: The strategies used for optimizing hierarchical design includes, top-down, bottom-up and compile-characterize. The advantages and disadvantages of each strategy are discussed in the subsequent section.

5. Setting Design Constraints: The constraints need to be set for the design optimization and for the timing analysis. The commands used in this step are

```
create_clock  
set_clock_skew  
set_input_delay  
set_output_delay  
set_max_area
```

6. Optimize Design: Synthesize the design to generate technology specific gate level netlist. The command used is

```
compile
```

7. Analyze and debug the design: This step is important to understand the potential problems in the design by generating various reports. The commands used in this step are

```
check_design  
report_area  
report_constraint  
report_timing
```

8. Generate Script file: The design database is stored in the form of script file.

Consider the top level object as full adder with inputs ‘a\_in, b\_in, c\_in’ and outputs ‘sum\_out, carry\_out’. The top-down compilation run is shown by using the following script and can be used in the practical scenario. Refer Chap. 10 for the sdc commands. To synthesize the design and compile use the script shown in Example 12.1.

```

/* read the design object */
read -format verilog full_adder.v
/* specify the technology requirements */
target_library = my_library.db
symbol_library = my_library.sdb
link_library = "*" + target_library
/* define the design environment */
set_load 2.0 sum_out
set_load 1.2 carry_out
set_driving_cell -cell FD1 all_inputs()
set_drive 0 clk_name
/* set the design constraints */
set_input_delay 1.25 -clock clk {a_in, b_in}
set_input_delay 3.0 -clock clk c_in
set_max_area 0
/* synthesize the design */
compile
/* generates reports */
report_constraint
report_area
/* save the design database */
write -format db -hierarchy -output full_adder.db

```

**Example 12.1** Key steps for synthesis and compilation [2]

## 12.2 Compilation Strategy

The methods used for compilation of any design can have top-down or bottom-up compilation approach. Each compilation method has its own advantages and disadvantages.

### 12.2.1 Top-Down Compilation

The top-down compilation uses the top level design constraints and is easier to execute as compared to the bottom-up compilation approach. Following are the advantages and disadvantages for the top-down compilation.

Advantages

1. Optimization engines work on full design, complete paths
2. Usually get best optimization result
3. No iteration required
4. Simpler constraints
5. Simpler data management

### Disadvantages

1. Longer runtime
2. More memory requirements
3. More runtime

The commands used for the top-down compilation are

```
dc_shell> current_design TOP
dc_shell> compile -timing_high_effort_script
```

### 12.2.2 Bottom-Up Compilation

The bottom-up compilation uses the submodule level compilation first and then it moves towards top level. Care must be taken by the designer to set “**set\_dont\_touch**” attribute on the submodules to avoid recompilation of the submodules. The designer needs to know the timing information for the inputs and outputs for each of the submodule. The advantages and disadvantages are discussed below

#### Advantages

1. Faster as compare to top-down compilation
2. Less processing required per run
3. Less memory requirement

#### Disadvantages

1. Optimization works on the submodule or subdesign
2. More iteration required
3. More hierarchies to maintain

Consider the design has two submodules. The commands used for the bottom-up compilation are

```
dc_shell> current_design submodule1
dc_shell> compile -timing_high_effort_script
dc_shell> set_dont_touch submodule1
dc_shell> current_design submodule2
dc_shell> compile -timing_high_effort_script
dc_shell> set_dont_touch submodule2
dc_shell> current_design TOP
dc_shell> compile -timing_high_effort_script
```

## 12.3 Area Minimization Techniques

There are several techniques used for minimizing the overall area of the design. The highest priority of the designer is to optimize for the timing followed by area. There are several efficient area minimization techniques at the RTL level. In the previous section, we have discussed about the resource sharing. Following are the key guidelines used to optimize for the area

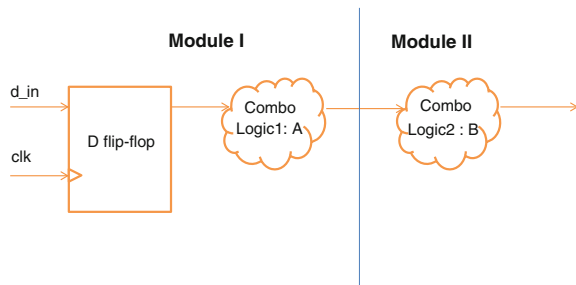
1. Avoid use of the combinational logic as individual block or module
2. Do not use the glue logic between two modules
3. Use `set_max_area` attribute while synthesizing the design.

### 12.3.1 Avoid Use of Combinational Logic as Individual Block

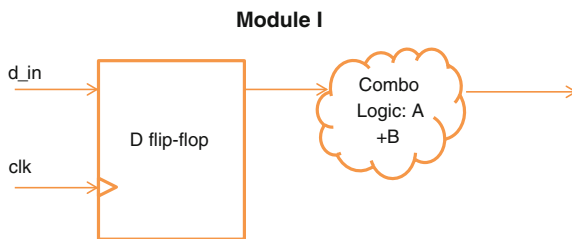
It is recommended not to use the combinational logic as individual block. If the individual combinational module is used then design compiler will not be able to optimize the individual block. This is not a good design partitioning. The hierarchy of the module is fixed and design compiler will not be able to modify the hierarchy of the design. Consider the scenario shown in Fig. 12.3. It has module I and module II, module II is individual combinational block so the design compiler will not be able to optimize module II, as design compiler does not optimize the port interfaces.

If the design is partitioned properly then the overall optimization will boost the design performance. A better partitioned ASIC design should have combined functionality of module I and module II. The functionality of  $A + B$  in the single module I is shown in Fig. 12.4 and results into faster optimization for the design.

**Fig. 12.3** Combinational logic as individual module



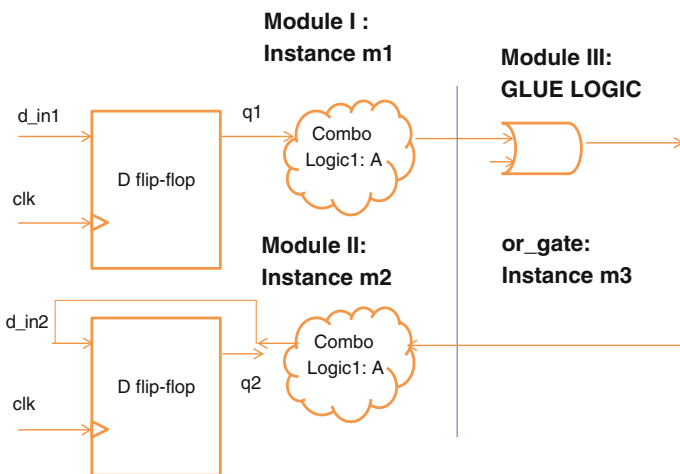
**Fig. 12.4** Eliminating individual combinational module



### 12.3.2 Avoid Use of Glue Logic Between Two Modules

If the module II in Fig. 12.3 is replaced by glue logic, that is instance of logic gate, then it glues between the different modules and is shown in Fig. 12.5. Such type of design partitioning is not good, the reason being the logic gate cannot be optimized by the design compiler and as design is not partitioned properly. To avoid this type of scenario it is recommended to use the *group* command. Either group the glue logic in the module I or module II. Following command is used to group the glue logic into module I:

```
dc_shell> group {m1, m3} -design_name moduleIII cell_name_or_gate
```



**Fig. 12.5** Glue logic between two blocks



Following command is used to group the glue logic into module II:

```
dc_shell> group {m2, m2} -design_name moduleIII
                cell_name or_gate
```

### 12.3.3 Use of *set\_max\_area* Attribute

To obtain the minimum possible area it is recommended to use the attribute *set\_max\_area*. This attribute is effective in the optimization of the design. Design compiler gives the highest priority to the timing optimization. If timing is met then only the area optimization phase can start. The priorities for the design optimization are listed below.

1. Design rule constraints (DRC)
2. Timing
3. Power
4. Area

### 12.3.4 Area Report

The area report is generated by the design compiler using *report\_area* command. The sample area report is shown in Example 12.2. The area report for any design consists of the number of ports, nets, references. It also gives information about the combinational, sequential, and total cell area.

Number of ports:	3
Number of nets:	8
Number of cells:	7
Number of references:	2
Combinational area:	100.349998
Non combinational area:	125.440002
Net Interconnect area: undefined (Wire load has zero net area)	
Total cell area:	225.790009
Total area:	undefined

**Example 12.2** Area report [3]

## 12.4 Timing Optimization and Performance Improvement

During optimization the timing has highest priority as compared to power and area. During the first phase of optimization the design compiler checks for the design rule constraints (DRC) violations, then the timing violations and the power constraints, and finally the area constraints. This section discusses about the few timing optimization commands supported by the design compiler.

### 12.4.1 Design Compilation with ‘map\_effort high’

Most of the time design engineer uses the option as map\_effort medium while performing the synthesis. It is advisable that during synthesis of the first phase, designer can use the option as map\_effort high as it reduces the compilation time. If the design constraints are not met then the designer can go for the incremental compilation with the option as map\_effort high. This can improve the design performance by at least 5–10 %.

The sdc command is shown below.

```
dc_shell> compile -map_effort_high -incremental_mapping
```

### 12.4.2 Logical Flattening

The design hierarchy of the design can be broken by using logical flattening of the design. The option allows all the logic gates of design at the same level of hierarchy. This allows the compiler to have better performance and better area utilization for the design. If the hierarchical design is large then this option may not work out. If number of hierarchies in the design increases, then compiler will take large amount of time for the design optimization.

Use the following command to achieve the logical flattening for the design

```
dc_shell> ungroup -all -flatten  
dc_shell> compile -map_effort_high -incremental_mapping  
dc_shell> report_timing -path full -delay max -max_path 1 -nworst 1
```

### 12.4.3 Use of group\_path Command

The design performance can boost unto 10 % by using the map\_effort high option. But if timing is not met with the incremental compilation by using the design constraints then it is essential to group the critical timing paths and use the weight factor to boost the design performance. This command is useful to improve the timing performance. The command is shown below.

```
dc_shell> group_path -name critical1 -from <input_name> -to <output_name> -weight <weight factor>
```

Consider the design scenario which has the setup violation of 0.38 ns. The setup violation is the difference between the data required time and data arrival time. So the slack is negative and setup time is violated.

```
dc_shell> read -format Verilog combinational_design.v
dc_shell> create_clock -name clk -period 15
dc_shell> set_input_delay 3 -clock clk in_a
dc_shell> set_input_delay 3 -clock clk in_b
dc_shell> set_input_delay 3 -clock clk c_in
dc_shell> set_output_delay 3 -clock c_out
dc_shell> current_design = combinational_design
dc_shell> compile -map_effort medium
dc_shell> report_timing -path full -delay max -max_path 1 -nworst 1
```

After the design is synthesized successfully use the **report\_timing** command while performing the timing analysis. The timing report for the synthesized design can be obtained with the multiple options as shown in the above script and shown in Example 12.3.

To fix the setup violation and to boost the design performance the designer can use the group\_path with the weight factor. More the weight factor, more is the compilation time.

```
dc_shell> group_path -name critical1 -from c_in -to c_out -weight 8
dc_shell> compile -map_effort high -incremental mapping
dc_shell> report_timing -path full -delay max -max_path 1 -nworst 1
```

Point	Incr	Path
Startpoint: c_in (input port)		
Endpoint: c_out (output port)		
Path Group: clk		
Path Type: max		
input external delay	0.00	0.00 f
c_in (in)	0.00	0.00 f
U19/Z (AN2)	0.87	0.87 f
U18/Z (EO)	1.13	2.00 f
add_8/U1_1/CO (FA1A)	2.27	4.27 f
add_8/U1_2/CO (FA1A)	1.17	5.45 f
add_8/U1_3/CO (FA1A)	1.17	6.62 f
add_8/U1_4/CO (FA1A)	1.17	7.80 f
add_8/U1_5/CO (FA1A)	1.17	8.97 f
add_8/U1_6/CO (FA1A)	1.17	10.14 f
add_8/U1_7/CO (FA1A)	1.17	11.32 f
U2/Z (EO)	1.06	12.38 f
C_out (out)	0.00	12.38 f
data arrival time		12.38 f
clock clk (rising edge)	15.00	15.00
clock network delay (ideal)	0.00	15.00
output external delay	-3.00	12.00
data required time		12.00
Data required time		12.00
Data arrival time		-12.38
Slack (violated)		-0.38

**Example 12.3** Timing report with negative slack [4]

The above commands generate the timing report with positive slack and remove setup violation and are shown in Example 12.4.

As shown in the above timing report for the max analysis with the compile\_map high option and weight factor of 8 the slack is met.

#### 12.4.4 Submodule Characterizing

In the practical ASIC designs, the design can have multiple hierarchies. Consider that the TOP level design consists of submodules X, Y, Z. If independently these submodules are synthesized and optimized for the design constraints they might meet the timing requirements independently. When these submodules are instantiated in the higher level of hierarchy (TOP) then it may be possible that they do not meet the timing. The reason for this may be the glue logic used among the submodules X, Y, Z or the tight constraints at the top level hierarchy.

Startpoint: c_in (input port)		
Endpoint: c_out (output port)		
Path Group: max		
Path Type: max		
Point	Incr	Path
input external delay	0.00	0.00 f
c_in (in)	0.00	0.00 f
U19/Z (AN2)	0.87	0.87 f
U18/Z (EO)	1.13	2.00 r
add_8/U1_1/CO (FA1A)	2.27	4.27 f
add_8/U1_2/CO (FA1A)	1.17	5.45 f
add_8/U1_3/CO (FA1A)	1.17	6.62 r
add_8/U1_4/CO (FA1A)	1.17	7.80 f
add_8/U1_5/CO (FA1A)	1.19	8.99 r
add_8/U1_6/CO (FA1A)	1.15	10.14 f
add_8/U1_7/CO (FA1A)	0.79	10.93 f
U2/Z (EO)	1.06	11.99 f
C_out (out)	0.00	11.99 f
data arrival time		11.99 f
clock clk (rising edge)	15.00	15.00
clock network delay (ideal)	0.00	15.00
output external delay	-3.00	12.00
data required time		12.00
Data required time		12.00
Data arrival time		-11.99
Slack (met)		0.01

**Example 12.4** Timing report with the positive slack [4]

Under such circumstances, to meet the design constraints, it is advisable to use the **characterize** command. This command allows the capturing of the boundary conditions for the submodule based on the TOP level hierarchy environment. Each submodule can be compiled and characterized independently.

Following is the small script which can enable the characterization of the individual submodules. Consider the submodule X, Y, Z and instance names as I1, I2, and I3.

```

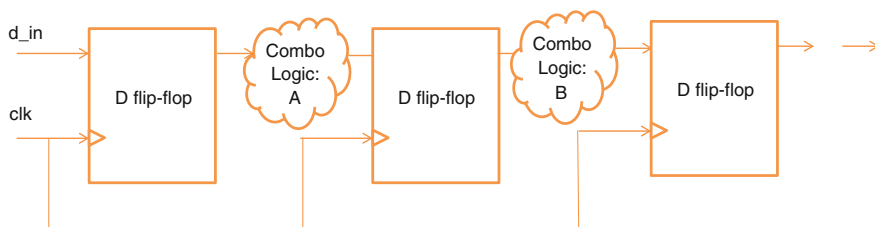
dc_shell> current_design=TOP
dc_shell> characterize I1
dc_shell> compile -map_effort high -incremental mapping
dc_shell> current_design=TOP
dc_shell> characterize I2
dc_shell> compile -map_effort high -incremental mapping
dc_shell> current_design=TOP
dc_shell> characterize I3
dc_shell> compile -map_effort high -incremental mapping
dc_shell> current_design=TOP

```

### 12.4.5 Register Balancing

Register balancing is very efficient and powerful command to move the combinational logic from one pipelined stage to another pipelined stage. This technique improves the design performance by moving the logic and hence reduces the register-to-register delay. Consider the pipelined design shown in Fig. 12.6 and consisting of the three flip-flops and combinational logic. In the first pipelined stage the combinational logic is 4-variable function and the second pipelined stage has combinational logic as 8-variable function and has more propagation delay as compare to the 4-variable combinational logic. Due to the different propagation delays in two different pipelined stages the design performance is based on the register-to-register timing path which has more delay.

Under such circumstances the register balancing can be used to shift the combinational logic from one of the pipelined stage to another pipelined stage without affecting the functionality of the design. This is achieved by compiler by using the following set of commands.



**Fig. 12.6** Pipelined stages

```
dc_shell> balance_registers
dc_shell> report_timing -path full -delay max -max_path 1 -nworst 1
```

### 12.4.6 FSM Optimization

For the optimization of the finite state machines the FSM Compiler is used. The use of FSM compiler is to achieve the small area optimization and to improve the design performance. In the practical ASIC designs the state machines are always coded as an independent block. The design which has state machines and the other logic cannot be considered as good design partitioning. The reason being, if the other required logic is isolated from the state machine logic then the designer can chose for the best suited encoding style while coding for the sate machines. So always use the different submodules for the logic and for the state machines for the better design performance.

Following script shown in Example 12.5 can be used for the FSM extraction and optimization.

```
/* read the design object */
dc_shell> read -format verilog state_machines.v
/* Map the design */
dc_shell> compile -map_effort medium
/* if the design is not partitioned then group the logic */
dc_shell> set_fsm_state_vector { <flip_flop_name>, <flip_flop_name>, ... }
dc_shell> group -fsm -design_name <fsm_design_name>
/* extract the state machine from netlist in the state machine table format */
dc_shell> set_fsm_state_vector { <flip_flop_name>, <flip_flop_name>, ... }
dc_shell> set_fsm_encoding { "state0=0", "state1=1", ..... }
dc_shell> extract
/* write the design in the FSM format */
dc_shell> write -format st -output state_machine.st
/* if the design is already in the state machine format then read the design */
dc_shell> read -format st state_machine.st
/* define the order of the state */
dc_shell> set_fsm_order {state0,state1,...}
/* define the encoding style */
dc_shell> set_fsm_encoding_style <encoding_style>
/* compile the design */
dc_shell> compile -map_effort high
```

**Example 12.5** FSM extraction script

### 12.4.7 Fixing Hold Violations

Fixing of the hold violations is very easy as compared to the setup violations. To fix the setup violations it is essential to modify the architecture of the design and in turn it has greater impact on the RTL coding of the design. The setup violations are fixed during the prelayout STA and hold violations can be fixed during pre layout or post layout STA phase. Design compiler is efficient to fix the hold violations automatically. Use the following command to fix the hold violation

```
dc_shell> set_fix_hold clk1
dc_shell> compile -map_effort_high- incremental_mapping
```

### 12.4.8 Report Command

Following are few commands used to generate reports.

#### 12.4.8.1 report\_qor

This is used to generate report which consists of timing summary of all the path groups. This gives overall status of the timing for the design. Example 12.6 shows the sample report with multiple timing path groups using *report\_qor* command.

#### 12.4.8.2 report\_constraints

This command is used to show the difference between the user constraints and the actual design values. Following is the sample Example 12.7 with *report\_constraints* command.

#### 12.4.8.3 report\_constraints\_all

This command is used to show all the timing and DRC violations. Following is the sample Example 12.8 by using the *report\_constraints\_all* command

```
max_delay/setup ('clk1' group)
```



Timing Path Group 'clk1'	
-----	
Levels of Logic:	6.00
Critical Path Length:	3.64
Critical Path Slack:	-2.64
Critical Path Clk Period:	11.32
Total Negative Slack:	-55.45
No. of Violating Paths:	59.00
No. of Hold Violations:	1.00
-----	
Timing Path Group 'clk2'	
-----	
Levels of Logic:	10.00
Critical Path Length:	3.59
Critical Path Slack:	-0.29
Critical Path Clk Period:	22.65
Total Negative Slack:	-2.90
No. of Violating Paths:	11.00
No. of Hold Violations:	0.00
-----	
Cell Count	
-----	
Hierarchical Cell Count:	1736
Hierarchical Port Count:	114870
Leaf Cell Count:	323324

**Example 12.6** Qor report

Weighted			
Group (max_delay/setup)	Cost	Weight	Cost
CLK	0.00	1.00	0.00
default	0.00	1.00	0.00
-----			
max_delay/setup	0.00		
Constraint	Cost		
-----			
max_transition	0.00 (MET)		
max_fanout	0.00 (MET)		
max_delay/setup	0.00 (MET)		
critical_range	0.00 (MET)		
min_delay/hold	0.40 (VIOLATED)		
max_leakage_power	6.00 (VIOLATED)		
max_dynamic_power	14.03 (VIOLATED)		
max_area	48.00 (VIOLATED)		
-----			

**Example 12.7** Report constraints

max_delay/setup ('clk1' group)			
	Required	Actual	
Endpoint	Path Delay	Path Delay	Slack
-----			
data[15]	1.00	3.64 f	-2.64 (VIOLATED)
data[13]	1.00	3.64 f	-2.64 (VIOLATED)
data[11]	1.00	3.63 f	-2.63 (VIOLATED)
data[12]	1.00	3.63 f	-2.63 (VIOLATED)
-----			

**Example 12.8** All constraint report

## 12.5 Constraint Validation

Following are the important commands used to validate the design (Table 12.1).

## 12.6 Commands for the DRC, Power, and Optimization

Following are the important commands used to define design rules, power, and optimization constraints (Table 12.2).

**Table 12.1** Constraint validation

Command [1]	Description
Check_design	Used to check for the design consistency and reports the unconnected nets, ports, etc.
Check_timing	Used to verify the timing setup is complete

**Table 12.2** DRC, power and optimization definition

Command [1]	Type	Description
set_max_transition	DRC	Used to define the largest transition time
set_max_fanout	DRC	Used to set the largest fanout for the design
set_max_capacitance	DRC	Used to set the maximum capacitance allowed for the design
set_min_capacitance	DRC	Used to set the minimum capacitance allowed for the design
set_operating_conditions	Optimization constraints	Used to set the PVT conditions as it affects on timing
set_load	Optimization constraints	Used to model load on output port
set_clock_uncertainty	Optimization constraints	Used to define the estimated network skew
set_clock_latency	Optimization constraints	Used to define the estimated source and network delays
set_clock_transition	Optimization constraints	Used to define the estimated input skew
set_max_dynamic_power	Power constraints	Used to set the maximum dynamic power
set_max_leakage_power	Power constraints	Used to set the maximum leakage power
set_max_total_power	Power constraints	Used to set the maximum total power
set_dont_touch	Optimization constraints	It is used to prevent the optimization of mapped gates

```

/* set the clock */
set clock clk
/* set clock period */
set clock_period 2
/* set the latency */
set latency 0.05
/* set clock skew */
set early_clock_skew [expr $ clock_period/10.0]
set late_clock_skew [expr $ clock_period/20.0]

/* set clock transition */
set clock_transition [expr $ clock_period/100.0]
/* set the external delay */
set external_delay [expr $ clock_period*0.4]
/* define the clock uncertainty*/
set_clock_uncertainty -setup $ early_clock_skew
set_clock_uncertainty -hold $ late_clock_skew

Name the above script as clock.src, and Source the above script

/* report clock and timing*/
dc_shell> report_timing
dc_shell> report_clock
dc_shell> report_timing
dc_shell> report_constraints -all_violations

```

**Example 12.9** Sample script for constraining design at 500 MHz

Example 12.9 is the sample script and can be used to constrain the design for operating frequency of 500 MHz.

## 12.7 Summary

Following are the key highlights to summarize this chapter

1. Constraints are classified as optimization, design rule, and environmental
2. Avoid use of the combinational logic as individual block or module
3. Do not use the glue logic between two modules
4. Use set\_max\_area attribute while synthesizing the design
5. The top-down compilation uses the top level design constraints and easier to execute as compare to the bottom-up compilation approach
6. The design hierarchy of the design can be broken by using logical flattening of the design.
7. Register balancing is very efficient and powerful command to move the combinational logic from one pipelined stage to another pipelined stage

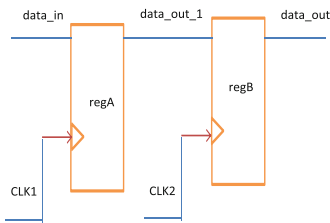
8. For the optimization of the finite state machines the FSM compiler is used. The use of FSM compiler is to achieve the small area optimization and to improve the design performance.

## References

1. [www.synopsys.com](http://www.synopsys.com) Design compiler® user guide version D-2010.03-SP2, June 2010
2. User guide version D-2010.03, March 2010
3. [www.synopsys.com](http://www.synopsys.com) Guidelines and practices for successful logic synthesis version 1998.08, August 1998
4. [www.synopsys.com](http://www.synopsys.com) Synopsys timing constraints and optimization

# Chapter 13

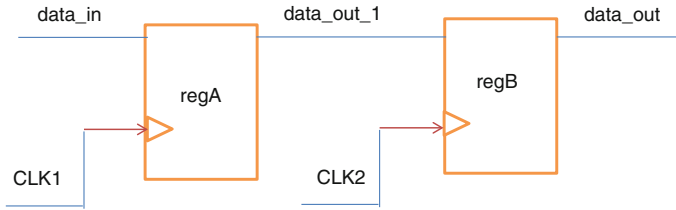
## Multiple Clock Domain Design



Multiple clock domain design understanding is essential for ASIC design engineer. In most of the practical design scenario the multiple clock domain designs are used and it is essential to understand and validate the legal converging states of the signals passing from one of the clock domain to another clock domain. This chapter is focused on the understanding of the multiple clock domain design techniques.

**Abstract** In the practical ASIC and SOC designs the multiple clocks are used and the designs are called as multiple clock domain designs. These kinds of designs need to be described using the efficient design architectures and Verilog RTL. This chapter focuses in the key design techniques which are used to describe the multiple clock domain designs while passing data from one of the clock domain to other. The chapter key highlights are the detail description for the synchronizers, data path, and control path synchronization logic using the efficient Verilog RTL. This chapter also discusses on the key design challenges in the multiple clock domain designs and even this chapter focuses on the design guidelines to describe the efficient clock domain designs.

**Keywords** Metastability • CDC • Skew • STA • FIFO • Level synchronizer • Pulse synchronizer • Mux synchronizer • FSM • Sending clock domain • Receiver clock domain • Edge detection • Level-to-pulse conversion • Gray counter • Binary to gray • Gray to binary • MCP • Convergence of data • Legal states • Gray encoding • FSM encoding • Reset synchronization • Data correlation • Setup and hold time



**Fig. 13.1** Multiple clock domain logic

### 13.1 What Is Multiple Clock Domain?

It is a very simple approach to design single clock domain design logic. If all the flip-flops in the design are clocked by single clock source then the design is said to be synchronous. If the flip-flops are triggered by the different clock sources then the design is said to be asynchronous clock domain design. In the modern ASIC or SOCs the design can have multiple clock sources of different frequencies. For example, consider Fig. 13.1, in this example the flip-flop regA is triggered by CLK1 and flip-flop regB is triggered by CLK2.

In Fig. 13.1 the data is sampled in the clock domain with the clock source CLK1 and output from the clock domain1 is data\_out\_1. The flip-flop is named as regA in clock domain1 and named as regB in the clock domain2. The regB has clock input as CLK2 and samples the data generated by clock domain1 on the rising edge of CLK2. The output from clock domain2 is data\_out. The difference between the single clock domain and multiple clock domain designs is the phase difference between arrivals of the clock signals. The clock sources CLK1 and CLK2 are different for both the domains and regardless of the same or different frequencies the design is treated as multiple clock domain design. The data is launched from one clock domain and captured in another clock domain.

### 13.2 What Is Clock Domain Crossing (CDC)

The data transfer can be from slower clock domain to faster clock domain or from faster clock domain to slower clock domain. The data or control signal crosses from one of the clock domains to another clock domain and it is described by the term clock domain crossing.

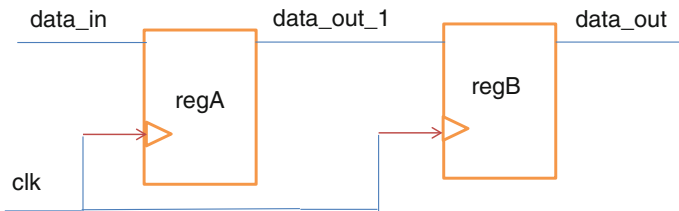
The synthesizable outcome of the Example 13.1 is shown in Fig. 13.2.

The timing sequence for Example 13.1 is shown in Fig. 13.3. In this the flip-flop propagation delays are not considered.

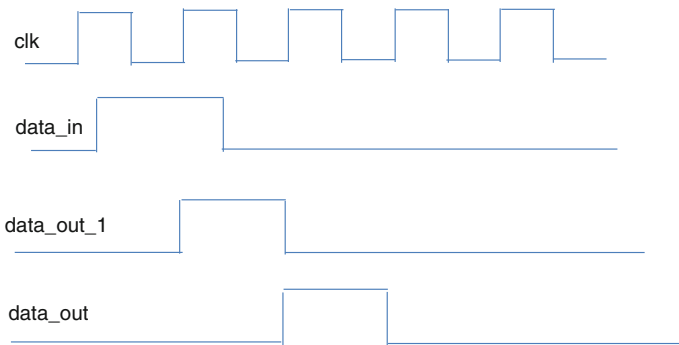
If we consider the multiple clock domain design described by Example 13.2, then there is an issue in the data integrity of the design due to unstable output or meta-stable output. In the practical use multiple clocks are not used in the same module.

```
module ( data_in, clk, data_out);  
input data_in;  
input clk;  
output reg data_out;  
reg data_out1;  
always@( posedge clk)  
  
begin  
data_out_1<= data_in;  
data_out <= data_out_1;  
end  
endmodule  
  
// the above Verilog code uses the single clock source CLK1 and generates the output data_out.
```

**Example 13.1** Verilog RTL for single clock domain design



**Fig. 13.2** Two-stage level synchronizer



**Fig. 13.3** Timing sequence for two-stage level synchronizer

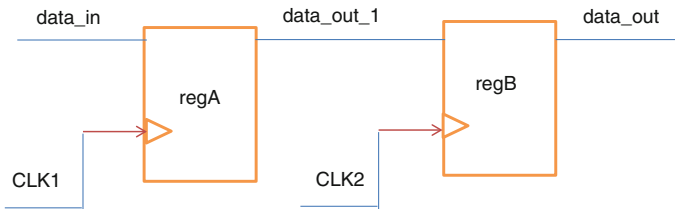


```

module ( data_in, clk1,clk2, data_out);
input data_in;
input clk1, clk2;
output reg data_out;
reg data_out1;
always@(posedge clk1)
begin
data_out_1<= data_in;
end
always@(posedge clk2)
begin
data_out<= data_out_1;
endmodule
// the above Verilog RTL code describes the multiple clock domain design scenario

```

**Example 13.2** Verilog RTL for multiple clock domain design



**Fig. 13.4** Synthesis result for the multiple clock domain design logic

Describe design functionality separately for different clock domains. That is module 1 can use clock source as CLK1 and module 2 can use clock source as CLK2.

The synthesizable outcome of the above Verilog RTL code is shown in Fig. 13.4.

The timing sequence for the Example 13.2 is shown in Fig. 13.5.

As shown in Fig. 13.5 the output from regB, i.e., data\_out is in the metastable state for one clock cycle. Metastability is the scenario in the design due to occurrences of multiple events very close to each other and due to that the setup and hold time violation occurs. The scenario results into the synchronization failure between multiple clock domain designs. It is due to different clock frequencies and different phases of the clock in the design. It is essential for the designer to think that why

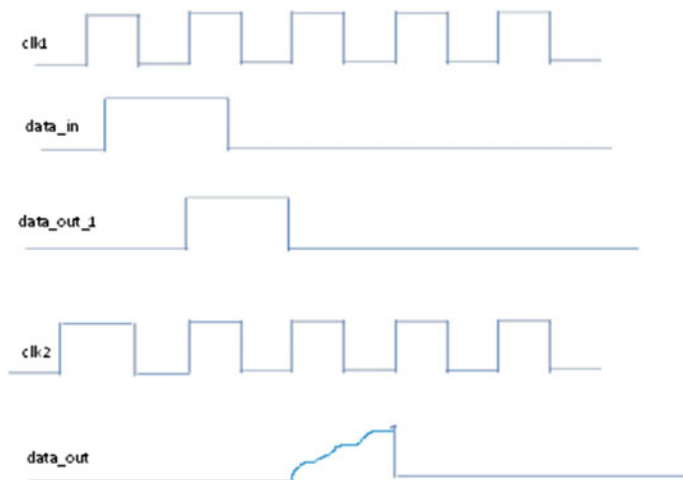


Fig. 13.5 Timing sequence for the metastable output

design goes into metastable state? The reasons being every flip-flop has setup and hold time and if the data changes during the setup time and hold time window then the design has timing violations and results into the unstable output. Metastable state of the design is not a stable state of the design so if the data output data\_out is fed to the other logic then the output from the other logic is unpredictable state or invalid logic state. So to avoid the metastable issues in the design it is essential to have synchronizers in the data path and control path for the design.

The issue of metastability can be resolved by adding the level synchronizers while sampling the control signals from one clock domain to another clock domain. Figure 13.6 shows the multiple clock domain design with the two-stage level-synchronizer logic.

As shown in Fig. 13.6 the level synchronizer is used in the second clock domain. The level synchronizer is designed using regC, regB and used to sample the data 'data\_out\_1' from the clock domain1. The register regC output can be metastable but register regB generates the stable legal output 'data\_out' on the next clock edge. The Verilog RTL is shown in Example 13.3.

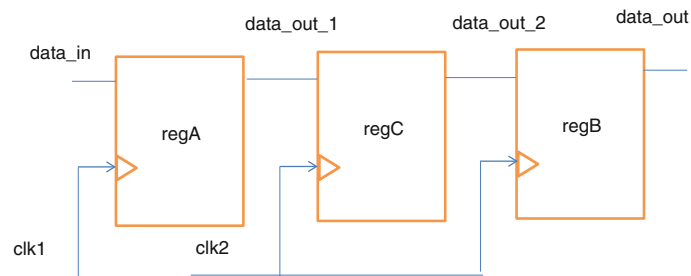
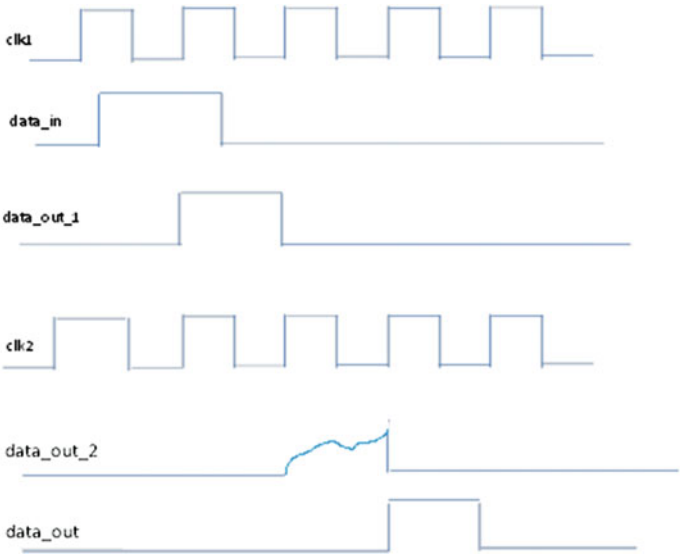


Fig. 13.6 Two-stage level synchronizer in the control path

```
module ( data_in, clk1,clk2, data_out);  
input data_in;  
input clk1, clk2;  
output reg data_out;  
reg data_out1, data_out_2;  
always@( posedge clk1)  
begin  
data_out_1<= data_in;  
end  
always@(posedge clk2)  
begin  
data_out_2<=data_out_1;  
data_out<= data_out_2;  
endmodule
```

**Example 13.3** Verilog RTL for the use of two-stage level synchronizer in the control path



**Fig. 13.7** Timing sequence with the use of two-stage synchronizer

The timing sequence for the same is shown in Fig. 13.7.

As shown in Fig. 13.7, during second rising edge of 'clk1' the output 'data\_out\_1' is assigned to data\_in and the register in the first clock domain generates an output logic '1'. On the third clock edge of 'clk2' the output of register 'regC', i.e., 'data\_out\_2' goes to the metastable state due to violation of either setup or hold time. But the register 'regB' timing parameters meet the convergence to sample the legal data value of logic '1' on the fourth clock edge of 'clk2'. Hence the output 'data\_out' goes to valid or legal value state of logic '1'. In most of the scenarios it is true that the output of 'regC', i.e., 'data\_out\_2' is in the metastable state due to violation of the setup and hold time. So during the timing analysis it is essential to set the 'false path' from output of 'regA' to the output of 'regC'. The false path is from regA/data\_out\_1 to regC/data\_out\_2. The SDC command for setting the false path is

```
set_false_path -from regA/q -to regC/q
```

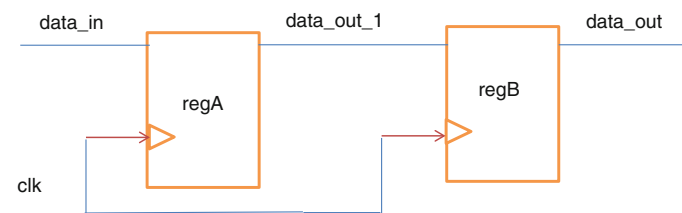
### 13.3 Level Synchronizers

The level synchronizers are used to pass the control signal information from one of the clock domains to another clock domain. In the practical scenario either two-stage or three-stage synchronizers are used. In the two-stage level synchronizer the number of registers used are two and three-stage level synchronizers are designed using three registers. The latency of control information transfer is dependent on the number of registers. The two-stage level synchronizer is shown in Fig. 13.8.

As described in the above section the functionality for the two-stage level synchronizer can be described using Example 13.4.

The three-stage level synchronizer is described using Verilog RTL as shown below and the synthesized logic is described in the Example 13.5 (Fig. 13.9).

In the multiple clock domain designs the data can be passed from slower clock domain to fast clock domain or from faster clock domain to slow clock domain depending on the design architecture and requirement. In both the cases the synchronizers need to be incorporated in the design. The synchronizers need to be incorporated in the data and control path for the design.



**Fig. 13.8** Level synchronizer logic diagram

```

always@(posedge clk)
begin
data_out_1<=data_in;
data_out<=data_out_1;
end

```

**Example 13.4** Verilog functional description for two-stage synchronizer

```

always@(posedge clk)
begin
data_out_1 <= data_in;
data_out_2 <= data_out_1;
data_out<= data_out_2;
end

```

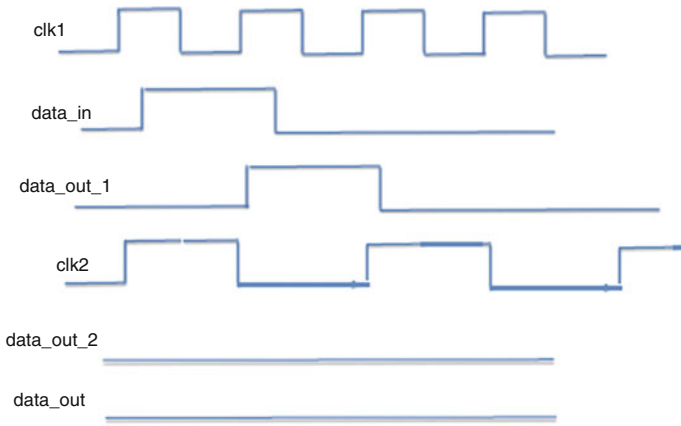
**Example 13.5** Verilog functional description for three-stage level synchronizer



**Fig. 13.9** Synthesis result for the three-stage level synchronizer

Passing of the control signal from the slower clock domain to the faster clock domain is not a problem as the signal launched by the slower clock domain can be sampled by the faster clock domain multiple times using the two-stage or three-stage level synchronizer.

As discussed above, consider 'clk1' is of 100 MHz and 'clk2' is of 200 MHz. As second clock domain is faster compared to the first clock domain, there is no any issue while sampling the control signals passed to the second clock domain. But in the practical design scenario problem occurs when the control information need to be passed from faster clock domain to the slower clock domain. The issue is due to nonconverging of the legal states of the control signals passed from clock domain1 to clock domain2.

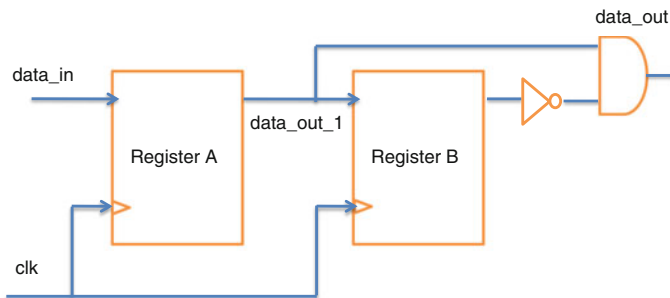


**Fig. 13.10** Timing sequence for capturing the data in the slower clock domain

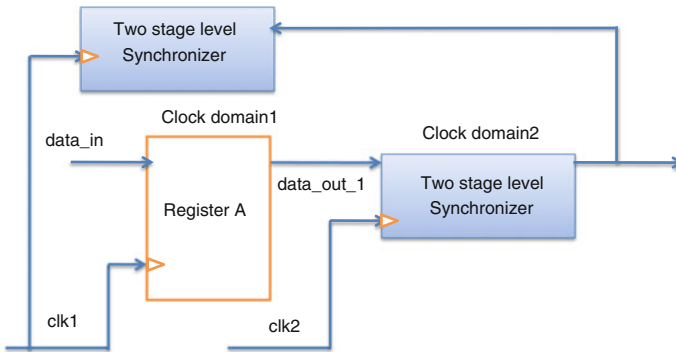
As shown in Fig. 13.10 due to slower clock ‘clk2’ in the clock domain2, the data output ‘data\_out\_1’ is sampled on the active edge of clock ‘clk2’ but unable to produce the desired output. Due to that both the registers in the second clock domain generate output logic ‘0’ and which is unexpected output. Both the ‘data\_out\_2’ and ‘data\_out’ outputs from the registers are at logic ‘0’ and shown in the timing sequence. The issue of sampling the data from faster clock domain to the slower clock domain can be resolved using pulse stretcher. The level-to-pulse generator on the positive clock edge is shown in Fig. 13.11.

Another mechanism to achieve the legal converging of the data is to use a handshaking mechanism using the handshaking signals.

As shown in Fig. 13.12 the sampled signal in the clock domain2 is reported as a handshaking signal to clock domain1. This handshake mechanism is like acknowledgment or notification to the faster clock domain1 that the control signal passed by the faster clock domain is successfully sampled by the slower clock domain. In most of the practical scenarios this kind of mechanism is used and even the faster clock domain can send another control signal after receiving the valid notification or acknowledgment signal from the slower clock domain.



**Fig. 13.11** Level-to-pulse conversion logic

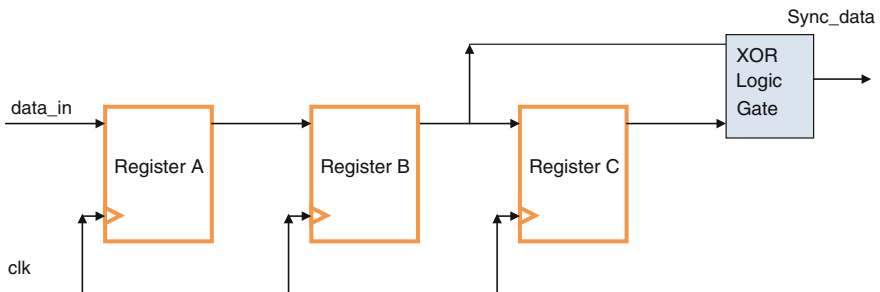


**Fig. 13.12** Handshake control signal mechanism

After receiving the valid handshake signal from clock domain 2 the output of two stage level synchronizer in the clock domain 1 should generate the output to control the next data\_in. This requires additional logic.

### 13.4 Pulse Synchronizers

This type of synchronizer uses the two-stage level synchronizer with the additional register to sample the output of two-stage level synchronizer. The output synchronized data is generated by XORing the output from the two-stage level synchronizer and the sampled output from the three-stage synchronizer. This kind of synchronizer is also named as toggle synchronizer and used to synchronize the pulse generated in sending clock domain into the destination clock domain. While passing the data from faster clock domain to the slower clock domain the pulse can be skipped if two-stage level synchronizer is used. In such scenarios the pulse synchronizers are very efficient and useful. The pulse synchronizer diagram is shown in Fig. 13.13.



**Fig. 13.13** Pulse synchronizer

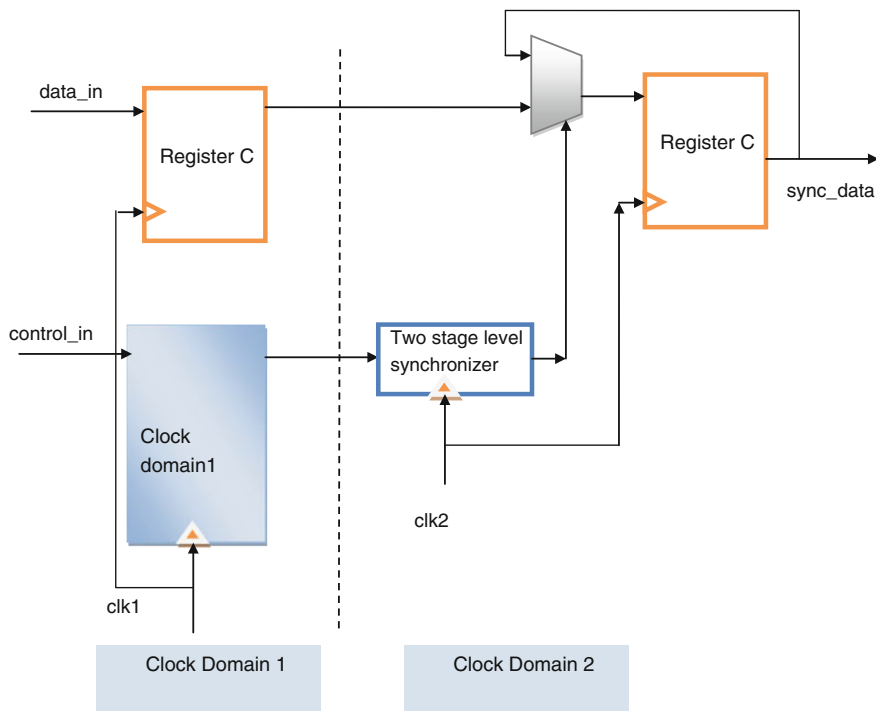


Fig. 13.14 Mux Synchronization

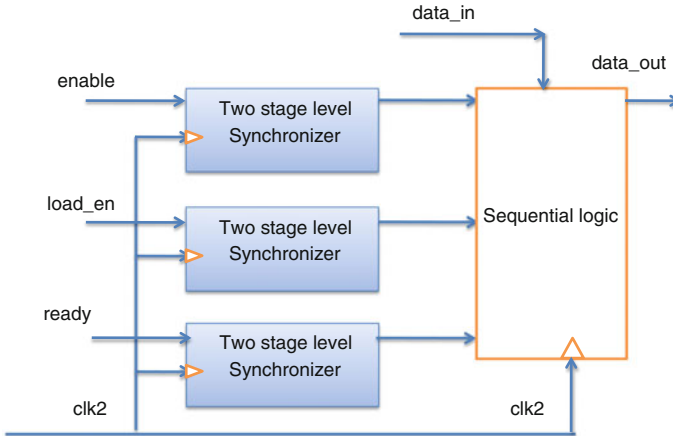
### 13.5 MUX Synchronizer

Use the pair of the data and control signals while sending the information from clock domain1 to clock domain2. Use the multiple-bit data and use the single-bit control signal. At the receiving end depending on the ratio of the sending clock and receiving clock use the level or pulse synchronizer to generate the control signal for the multiplexer. This technique is similar to the MCP and effective if the data is stable for multiple clock cycles across the clock boundaries. The diagram is shown in Fig. 13.14.

### 13.6 Challenges in the Design of Synchronizers

Passing multiple control signals from one of the clock domains to another clock domain is one of the key challenges for an ASIC or SOC design engineer. When multiple signals are passed from one of the clock domains to another clock domain then the arrival time of the entire control signals is very important. If all the control signals are arrived at a time then the skew is zero. Then there is no issue while capturing these signals in another clock domain. But in the practical scenario, it





**Fig. 13.15** Sampling multiple signals in the receiver clock domain

may be possible that there may be skew between the multiple control signals due to different arrival time from clock domain1 to clock domain2. And this can be the cause of the synchronization failure. Consider the design scenario shown in Fig. 13.15, where ‘enable’, ‘load\_en’, and ‘ready’ need to be passed from one of the clock domains to another clock domain. In such scenario, if independent-level synchronizers are used for all the required control signals then there might be synchronization failure at the receiving end due to skew.

Consider the case where ‘ready’ and ‘load\_en’ are arrived and sampled at a time but due to late arrival of ‘enable’ input in the receiving clock domain2. The data output from the first register of synchronizer does not change and it does not sample the new value. Hence there is practical issue in generating valid legal state output. The sampling of multiple control signals is described in the Example 13.6.

The practical and feasible design solution to resolve this problem is to develop the logic in the clock domain1 to generate the single control signal using ‘enable’, ‘load\_en’, and ‘ready’. Pass this single-bit control signal from clock domain1 to clock domain2. The architecture modification is shown in Fig. 13.16.

The Verilog RTL is shown in Example 13.7 for the receiving second clock domain logic.

### Design Scenario I

Consider the design scenario for passing the multiple signals from clock domain1 to clock domain2. If clock domain1 generates two output signals ‘enable\_1’ and ‘enable\_2’ and the receiving clock domain2 uses these two signals for the pipelined control logic as illustrated in the Example 13.8 then there might be a chance of synchronization failure. The synthesized logic is shown in Fig. 13.17.

The issue for the Verilog RTL described in Example 13.8 is sampling of ‘enable\_1’ and ‘enable\_2’ in the receiving clock domain2. Although the two-level synchronizers are used to sample ‘enable\_1’ and ‘enable\_2’ the small skew

```

always@(posedge clk2)
begin
load_en_c2_tmp<= load_en_c1;

load_en_c2 <= load_en_c2_tmp;

end

always@(posedge clk2)
begin
enable_c2_tmp<=enable_c1;

enable_c2<=enable_c2_tmp;

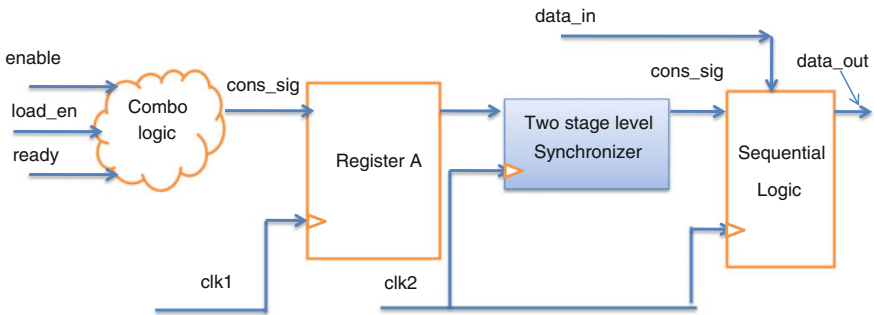
end

always@(posedge clk2)
begin
ready_c2_tmp<= ready_c1;

ready_c2 <= ready_c2_tmp;

end
    
```

**Example 13.6** Verilog functionality to sample the multiple control signals



**Fig. 13.16** Consolidated control signal passing in the multiple clock domains

between arrivals of 'enable\_1' and 'enable\_2' can cause the issue. The pipelined stage shown in Fig. 13.17 can miss the data due to this issue and can result in invalid output.

```

always@(posedge clk2)

begin

control_signal_tmp<= control_input;

control_singnal_c2 <= control_signal_tmp;

end

always@(control_signal_c2)

begin

{enable, ready, load_en} = {3 { control_signa_c2}};

end

```

**Example 13.7** Partial Verilog RTL for consolidated control signal receiving

Figure 13.18 illustrates that the ‘data\_out’ is permanently zero and not loaded due to the skew between the ‘enable1\_1 and ‘enable2\_1. If these two signals have skew then there is a gap of clock cycle while sampling these signals in the receiving clock domain.

**Solution** The practical solution is to use the consolidated enable signal and sample ‘enable\_cons’ in the second clock domain to generate the valid ‘enable2\_2’ signal from the output of two-stage level synchronizer. Figure 13.19 shows the generation and use of the consolidated control signal (Example 13.9).

## Design Scenario II

Consider the design scenario of passing the multiple-bit encoder output from one of the clock domains to another clock domain. Consider that encoder outputs ‘encoder\_1’, ‘encoder\_2’ are passed from the clock domain1 to clock domain2. The output generated by the clock domain1 is sampled by the clock domain2 using the two-stage level synchronizer. The output of level synchronizer is used as an input to 2:4 decoder. There might be a chance that the decoder output is error prone if there is skew between the inputs ‘encoder\_1’ and ‘encoder\_2’.

The Verilog RTL functionality in the clock domain2 is shown in the Example 13.10.

Consider the practical scenario with reference to Example 13.10, the issue in the output is due to skew between ‘encoder\_1’ and ‘encoder\_2’. Due to the skew the decoder output ‘decoder\_out[1]’ is permanently zero and never be asserted. This problem can be fixed using the enable control signal while sampling the ‘encoder\_1’ and ‘encoder\_2’ signals from the clock domain1 by clock domain2. The enable control signal can be of one clock duration wide and can act as device ready or control signal to pass the control information when ‘enable=1’. The enable signal

```
module multiple_clock_signals ( clk2, enable_1,enable_2, data_in, data_out);  
  
input enable_1,enable_2;  
  
input data_in;  
  
input clk2;  
  
output reg data_out;  
  
reg data_out_2;  
reg enable_1to2, enable_1to2_1, enable_1to2_tmp, enable_1to2_tmp1,  
enable_1to2_tmp2;  
  
always@(posedge clk2)  
  
begin  
  
enable_1to2_tmp <= enable_1;  
  
enable_1to2 <= enable_1to2_tmp;  
  
end  
  
always@(posedge clk2)  
  
begin  
  
enable_1to2_tmp1 <= enable_2;  
  
enable_1to2_1 <= enable_1to2_tmp1;  
  
end  
  
always@(posedge clk2)  
  
begin  
  
data_out_2 <= data_in && enable_1to2;  
  
data_out <= data_out_2 && enable_1to2_1;  
  
end
```

**Example 13.8** Verilog RTL for using multiple signals for pipelined operation

can be asserted while asserting the encoder output or enable signal can be asserted after one clock cycle after assertion of the encoder output. Assertion and deassertion logic can be designed separately for enable input.

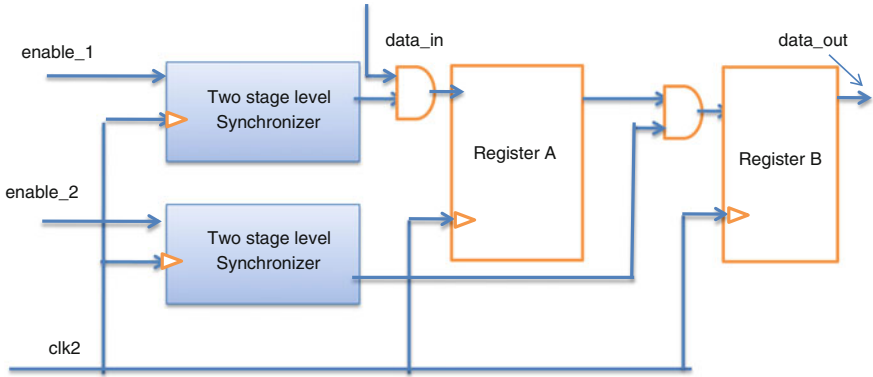


Fig. 13.17 Passing of multiple signals for the pipelined operation

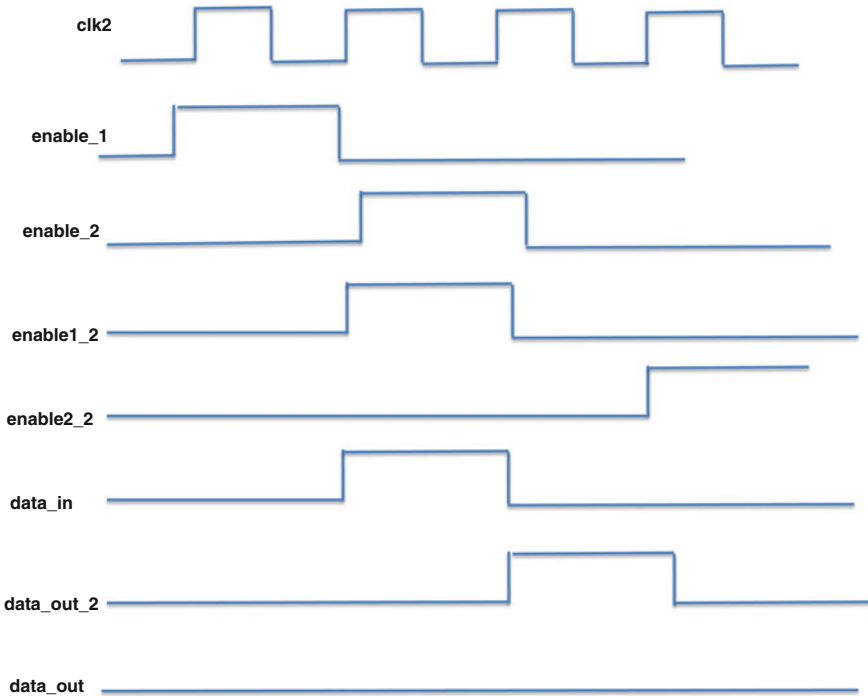
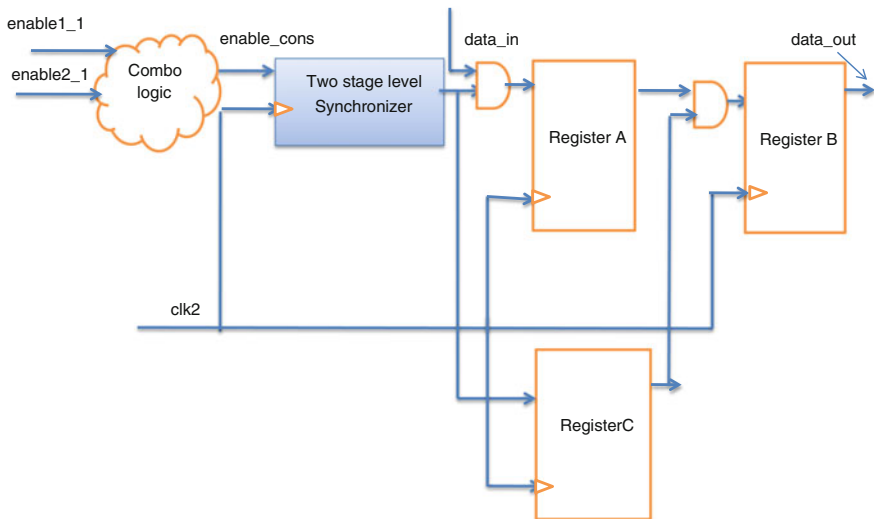


Fig. 13.18 Timing sequence for the use of multiple control signals for pipelined control logic

Another important practical and viable approach is to generate the decoder output in the clock domain1 itself by repartition of the design and sample the decoder output in the clock domain2 using the consolidated enable input and two-level synchronizers.



**Fig. 13.19** Modified architecture to register the consolidated control signal for pipelined logic

```

always@(posedge clk2)
begin
{enable_2, enable_2_tmp} <= { enable_2_tmp, enable_cons};
enable_2_2 <= enable_2;
end
always@(posedge clk2)
begin
data_out_2 <= data_in && enable_2;
data_out <= data_out_2 && enable_2_2;
end
    
```

**Example 13.9** Partial Verilog RTL for the use of the consolidated control signals for pipelined logic

The following Verilog RTL describes the sampling of the decoder output in the clock domain2 (Example 13.11).

```

always@(posedge clk2)
begin
{encoder1_2,encoder1_2_tmp} <= { encoder_1_tmp, encoder_1};
{encoder2_2,encoder2_2_tmp} <= { encoder_2_tmp, encoder_2};
end
always@(encoder1_2, encoder2_2)
begin
case { encoder1_2,encoder2_2}
2'b00 : decoder_out =4'b1110;
2'b01 : decoder_out = 4'b1101;
2'b10 : decoder_out = 4'b1011;
2'b11 : decoder_out =4'b0111;
endcase
end

```

**Example 13.10** Partial Verilog RTL for the sampling of encoder output

## 13.7 Data Path Synchronizers

As discussed in the above section to pass the multi-bit signals from one of the clock domains to another clock domain is difficult and error-prone task. Although the multistage level synchronizers can be used due to skew between the multiple clock signals the synchronization cannot be achieved. So for the multi-bit data the other techniques are used to pass the data from one of the clock domains to another clock domain. There are two main techniques to pass multi-bit data and these are used in the practical ASIC designs. These techniques are

- (a) Handshaking mechanism
- (b) FIFO memory buffers

### 13.7.1 Handshaking Mechanism

As discussed in the previous section, one or more than one handshake signals are required while passing the data from one of the clock domains to another clock

```

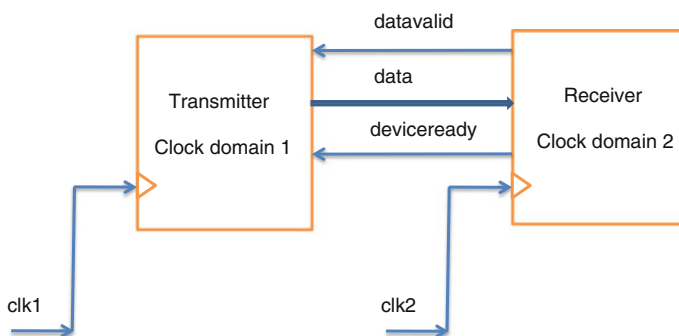
always@(posedge clk1)
begin
case { encoder_1,encoder_2}
2'b00 : decoder_out =4'b1110;
2'b01 : decoder_out = 4'b1101;
2'b10 : decoder_out = 4'b1011;
2'b11 : decoder_out =4'b0111;
endcase
end

always@(posedge clk2)
begin
{decoder_out_2, decoder_out_tmp} <= ( decoder_out_tmp, decoder_out);
end

```

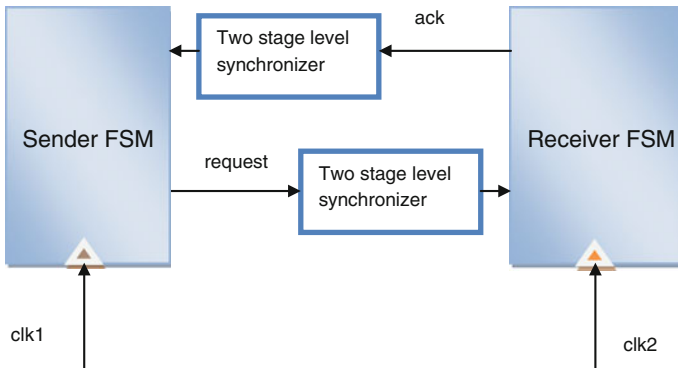
**Example 13.11** Partial Verilog RTL for the pushing decoder in the single clock domain

domain. Consider the design scenario shown in Fig. 13.20, where the multi-bit data need to be passed from the transmitter to receiver. The transmitter is clocked in the clock domain1 and receiver is clocked by another clock in the second clock domain. So the multi-bit data exchange using only level synchronizer is not effective while



**Fig. 13.20** Block diagram for handshake mechanism





**Fig. 13.21** FSM handshaking mechanism

passing data from transmitter to receiver. ASIC designer can think of the architecture by incorporating the handshake signals ‘datavalid’ and ‘deviceready’. In most of the practical scenario where latency is not a bottleneck this mechanism is effective to pass multi-bit data.

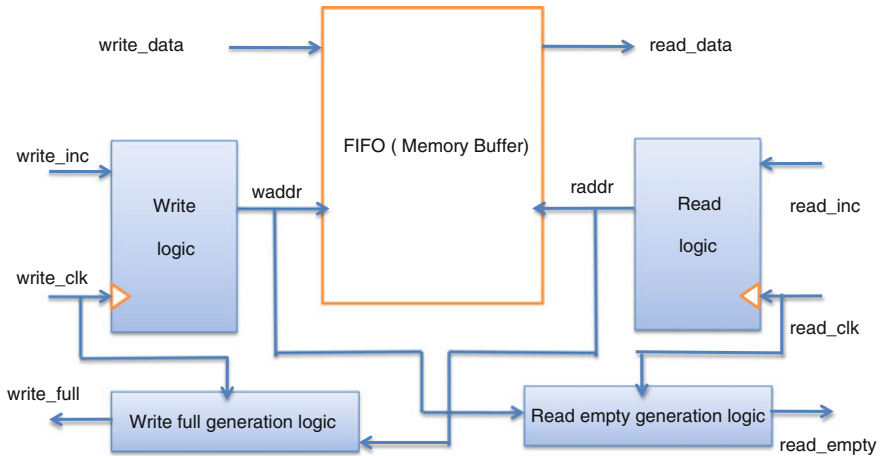
As shown in Fig. 13.20, when transmitter passes multi-bit data from clock domain1, then the receiver receives the data in the another clock domain using receive clock edge and generates active high ‘datavalid’ signal to indicate that the valid data has been received in the second clock domain. So the transmitter uses this signal ‘datavalid’ as handshaking signal. So until ‘datavalid’ signal is active high the transmitter cannot place the new data on the data lines. As two-or three-stage level synchronizers are used to sample the data in the second clock domain, it is recommended that the ‘datavalid’ signal should be active for at least two or three clock cycles. The overall latency while transferring the data is dependent upon the number of synchronizer stages and number of handshaking signals used. The poor latency is one of the biggest disadvantages of the handshake mechanism.

If required in most of the cases, another handshaking signal ‘deviceready’ can be generated with the ‘datavalid’ signal. The receiving clock domain can notify to the transmitter clock domain about the receiver status by asserting the ‘deviceready’ signal. But while designing handshake mechanism care needs to be taken for the generation of ‘deviceready’ and ‘datavalid’ signals. The ‘deviceready’ handshake signal should go to logic ‘1’ after deassertion of ‘datavalid’ signal.

For the FSM control use the architecture to establish the handshake across the clock domains using request and ack signals, Fig. 13.21.

### 13.7.2 FIFO Synchronizer

In the practical ASIC design scenario, the FIFO memory buffers are used as data path synchronizers to pass the data between multiple clock domains. The sender



**Fig. 13.22** Block diagram of FIFO

clock domain or transmitter clock domain can write the data into the FIFO memory buffer using `write_clk` and receiver clock domain can read the data using `read_clk`.

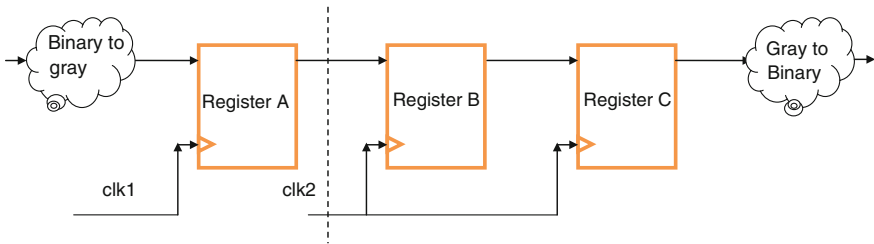
So basically FIFO consists of the memory buffer, write domain logic, read domain logic, and the empty and full flag generation logic. The FIFO with different logic blocks is shown in Fig. 13.22.

### 13.7.3 Gray Encoding

While passing the multiple bits of the data or control signals it is essential to use the gray encoding technique as it is guaranteed to sample the single-bit change in the receiving clock domain. For example, if 4-bit binary data need to be passed using the binary counter from sending clock domain to receiver clock domain then use the binary-to-gray converter logic in the sender clock domain. This guarantees only one-bit change across the clocking boundary. After sampling the gray counter value in the receiving clock domain use gray-to-binary conversion logic to perform the operations on the binary numbers. The technique is described in Fig. 13.23.

#### 13.7.3.1 Gray-to-Binary Converter

Please refer the Chap. 2 for the Verilog RTL of gray-to-binary converter. The gray-to-binary code conversion for 4-bit number is described in the Example 13.12.



**Fig. 13.23** Gray encoding technique

```

module gray_to_binary ( gray, binary);
parameter data_size =4;
input [data_size-1 :0] gray;
output reg [data_size-1:0] binary;
integer m;
always@(gray)
begin
for (m=0; m< data_size; m++)
binary[m] = ^ (gray >> m);
end
endmodule

```

**Example 13.12** Verilog RTL for the gray-to-binary converter

### 13.7.3.2 Binary-to-Gray Converter

Please refer the Chap. 2 for the Verilog RTL of binary-to-gray converter. The binary-to-gray code conversion for 4-bit number is described in the Example 13.13.

### 13.7.3.3 Practical Gray Code Counter

In the multiple clock domain designs it is recommended to use the Gray codes as in the two successive Gray numbers only one bit changes. The Verilog RTL for the Gray counter is shown in Example 13.14 (Fig. 13.24).

```
module binary_to_gray ( binary, gray);  
  
parameter data_size =4;  
  
input [data_size-1 :0] binary;  
  
output reg [data_size-1:0] gray;  
  
integer m;  
  
always@(binary)  
  
begin  
  
gray = (binary >>1) ^ binary;  
  
end  
  
endmodule
```

**Example 13.13** Verilog RTL for binary-to-gray converter

## 13.8 Design Guidelines for the Multiple Clock Domain Designs

CDC design errors can cause the serious design failures. These design failures are expensive during the chip design cycle. These design failures can be avoided using the following few guidelines during the design and verification phases.

1.

*Metastability* While passing the control signal information or data information, use the register logic in the sending clock domain. The reason being, if unregistered logic is used to pass the data from the sending clock domain to the receiver clock domain then there might be chances of glitches or hazards due to the multiple transitions in the single clock cycle. This can force the register logic into the metastable state due to violation of setup or hold time. The multiple transitions during single clock cycle can be avoided using the register logic while passing the data. Metastability blocking logic is shown in Fig. 13.25.

2.

*Use of MCP* Multi-Cycle Path formulation is highly recommended to avoid the metastability issue while passing the data and control signal information across the clock domains. In the MCP the strategy is to create the control and data pairs to pass the multi-bit data and single-bit control signal from sending clock domain to receiving clock domain. The control information can be sampled in the receiving clock domain using the pulse synchronizer and data can be passed to the receiving clock domain from sending clock domain with or without synchronizers. This technique is highly effective as the data can maintain the stable value for multiple

```

module gray_counter ( clk, increment, reset_n, gray);
parameter data_size =4;
input clk;

input reset_n;

input increment;

output reg [data_size-1:0] gray;

output reg [data_size-1:0] gray_next, binary_next, binary;

integer m;

always@(posedge clk or negedge reset_n)

if (!reset_n)

gray <= 4'b0000;

else

gray <= gray_next;

always@(gray, increment)

begin

for (m=0; m< data_size; m++)

binary[m] =^ (gray >>m);

binary_next = binary +increment;

gray_next = (binary_next >>1) ^ binary_next;

end

endmodule

```

**Example 13.14** Verilog RTL for the gray code counter

cycles and can be sampled in the receiving clock domain using the synchronized signal generated using pulse synchronizer. Across the clock domain crossing boundaries following are key points need to be considered:

- a. Control signals must be synchronized using the multistage synchronizers.
- b. Control signals should be free of hazards and glitches.
- c. There should be single transition across clock boundaries.
- d. Control signal should be stable for at least single clock cycle.

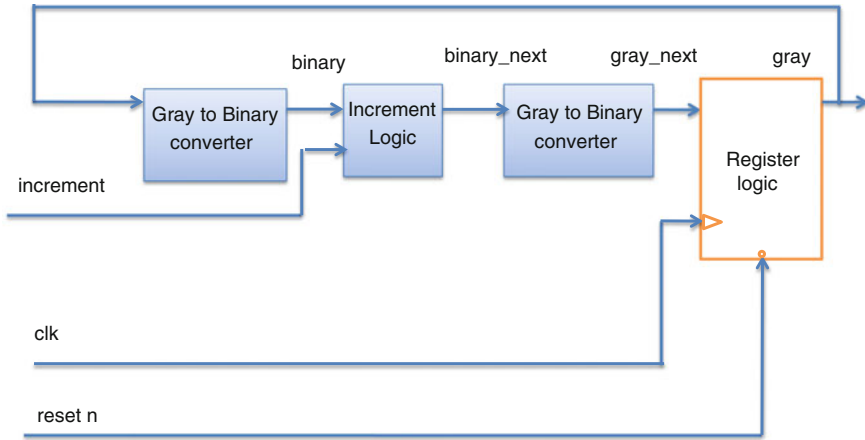


Fig. 13.24 Synthesis diagram for gray counter

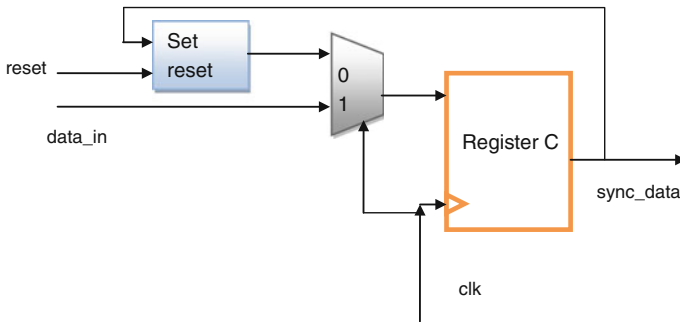


Fig. 13.25 Metastability blocking logic

The MCP formulation is shown in Fig. 13.26.

3. *Use FIFO* The common and effective technique to pass the multi-bit control or data information is the use of asynchronous FIFO. In this technique the sending clock domain writes the data into FIFO memory buffer and receiving clock domain reads the data from the FIFO buffer.
4. *Use gray code counters* In most of the ASIC designs with CDC, it is essential to pass the counter values across the clock domains. If binary counters are used across the clock domain boundaries then due to one or many bit change at a time the sampling at the receiver clock domain is difficult and error prone due to the multiple transitions. In such scenarios it is recommended to use the Gray code counters to pass the data across the clock boundaries (Example 13.14, Fig. 13.24).
5. *Design partitioning* While designing the logic for the multiple clock domain design, partition the design using the single clock for every individual module.

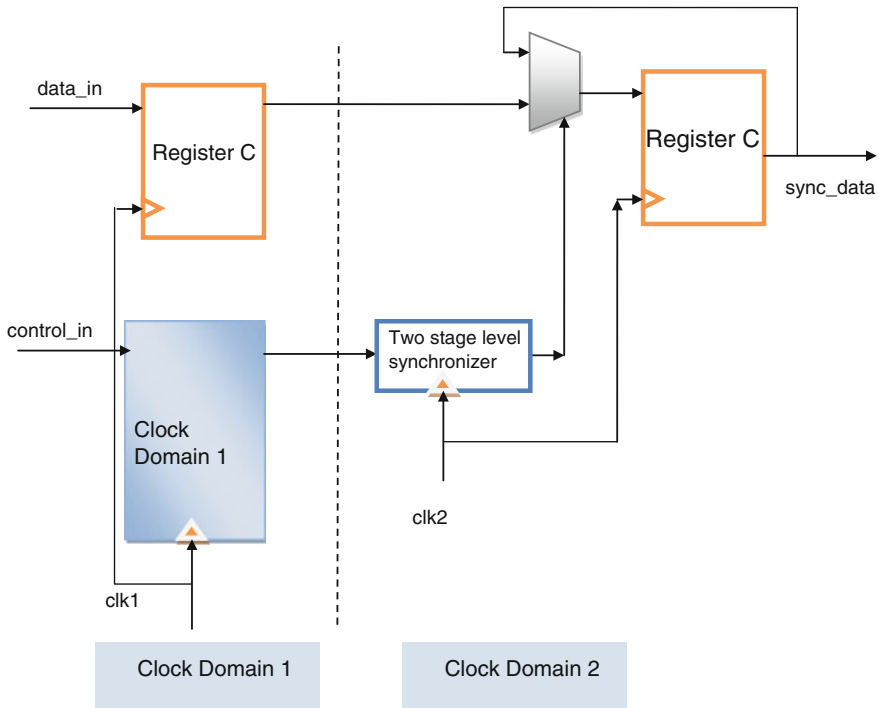


Fig. 13.26 MCP formulation

This is highly recommended as the STA will be easier due to clean reg-to-reg paths. Even the design verification will be easier due to the design partitioning and using the single clock.

6. *Clock naming conventions* It is recommended to use the clock naming conventions to identify the clock source. The naming conventions for the clock should be supported by the meaningful prefix. For example, for sending clock domain use `clk_s` and for the receiving clock domain use `clk_r`.
7. *Reset synchronization* For the ASIC and SOC designs it is highly recommended to use the reset synchronizers while asserting the reset and even it is essential to incorporate the reset synchronizer to avoid the metastability during reset de-assertion. Every SOC has single reset and either it is positive-level sensitive or negative-level sensitive. So if synchronizers are not used then there are chances of metastable state of flip-flops.
8. *Avoid hold time violations* To avoid the hold time violations it is recommended to pass the stable data for multiple clock cycles from the sending clock domain to the receiving clock domain. If data is passed from the faster clock domain to the slower clock domain then the data should be stable for multiple clock cycles to avoid the hold time violations.

9. *Avoid loss of correlation* Across the clock domain boundary there are several ways due to which loss of correlation can occur. Few of them are
- Multiple bits on the bus
  - Multiple handshake signals
  - Unrelated signals

To avoid this use the clock intent verification technique. This technique will ensure the passing of multi-bit signal across the clock boundaries.

## 13.9 FIFO Depth Calculations

### Asynchronous FIFO Depth Calculations:

**Scenario I:** Clock domain 1 is faster as compare to clock domain 2 that is  $f_1$  is greater than  $f_2$  without any idle cycle between write and read

Consider the design where  $f_1 = 100$  MHz and  $f_2 = 50$  MHz and the burst of data transfer from clock domain one to clock domain 2 is 100 without idle cycles that is consecutive write and read cycles.

The depth of FIFO can be calculated as :

1. **Find time required to write one data :**

$$T_{\text{write}} = 1/100 \text{ MHz} = 10 \text{ nsec}$$

2. **Find out time required to write burst of data :**

$$T_{\text{burst\_write}} = T_{\text{write}} * \text{Burst length} = 10\text{nsec} * 100 = 1\text{micro-second}$$

3. **Find time required to read one data :**

$$T_{\text{read}} = 1/50 \text{ MHz} = 20 \text{ nsec}$$

4. **Find out number of data read in duration of  $T_{\text{burst\_write}}$  :**

$$\text{No of reads} = 1000 \text{ nsec} / 20 \text{ nsec} = 50$$

5. **The depth of FIFO :**

$$\text{Depth} = \text{Burst length} - \text{No of reads} = 100 - 50 = 50$$



**Scenario II:** Clock domain I is faster as compare to clock domain 2 that is  $f_1$  is greater than  $f_2$  with idle cycles between writes and reads.

Consider the design where  $f_1 = 100$  MHz and  $f_2 = 50$  MHz and the burst of data transfer from clock domain one to clock domain 2 is 100 with idle cycles . Number of idle cycles between two successive writes = 1 and number of idle cycle between two successive reads =3

The depth of FIFO can be calculated as :

1. **Find time required to write one data :**

As between two successive writes the idle cycle is one therefore for every two cycles one data is written

$$T_{write} = 2 * (1/100 \text{ MHz}) = 20 \text{ nsec}$$

2. **Find out time required to write burst of data :**

$$T_{burst\_write} = T_{write} * \text{Burst length} = 20\text{nsec} * 100 = 2 \text{ micro-second}$$

3. **Find time required to read one data :**

As between two successive reads the idle cycle is three therefore for every four cycles one data is read

$$T_{read} = 4 * (1/50 \text{ MHz}) = 80 \text{ nsec}$$

4. **Find out number of data read in duration of  $T_{burst\_write}$  :**

$$\text{No of reads} = 2000 \text{ nsec} / 80 \text{ nsec} = 25$$

5. **The depth of FIFO :**

$$\text{Depth} = \text{Burst length} - \text{No of reads} = 100 - 25 = 75$$

**Scenario III:** Clock domain I is slower as compare to clock domain 2 that is  $f_1$  is less than  $f_2$  with idle cycles between two successive writes and two successive reads.

Consider the design where  $f_1 = 50$  MHz and  $f_2 = 80$  MHz and the burst of data transfer from clock domain one to clock domain 2 is 100 with idle cycles . Number of idle cycles between two successive writes = 1 and number of idle cycle between two successive reads =3

The depth of FIFO can be calculated as :

**1. Find time required to write one data :**

As between two successive writes the idle cycle is one therefore for every two cycles one data is written

$$T_{write} = 2 * (1/50 \text{ MHz}) = 40 \text{ nsec}$$

**2. Find out time required to write burst of data :**

$$T_{burst\_write} = T_{write} * \text{Burst length} = 40\text{nsec} * 100 = 4 \text{ micro-second}$$

**3. Find time required to read one data :**

As between two successive reads the idle cycle is three therefore for every four cycles one data is read

$$T_{read} = 4 * (1/80 \text{ MHz}) = 50 \text{ nsec}$$

**4. Find out number of data read in duration of  $T_{burst\_write}$  :**

$$\text{No of reads} = 4000 \text{ nsec} / 50 \text{ nsec} = 80$$

**5. The depth of FIFO :**

$$\text{Depth} = \text{Burst length} - \text{No of reads} = 100 - 80 = 20$$

**Scenario IV:** Clock domain I is frequency is equal to clock domain 2 that is  $f_1$  is equal to  $f_2$  and idle cycles between two successive reads and writes

Consider the design where  $f_1 = 100$  MHz and  $f_2 = 100$  MHz and the burst of data transfer from clock domain one to clock domain 2 is 100 with idle cycles . Number of idle cycles between two successive writes = 1 and number of idle cycle between two successive reads =3

The depth of FIFO can be calculated as :

**1. Find time required to write one data :**

As between two successive writes the idle cycle is one therefore for every two cycles one data is written

$$T_{write} = 2 * (1/100 \text{ MHz}) = 20 \text{ nsec}$$

**2. Find out time required to write burst of data :**

$$T_{burst\_write} = T_{write} * \text{Burst length} = 20\text{nsec} * 100 = 2 \text{ micro-second}$$

**3. Find time required to read one data :**

As between two successive reads the idle cycle is three therefore for every four cycles one data is read

$$T_{read} = 4 * (1/100 \text{ MHz}) = 40 \text{ nsec}$$

**4. Find out number of data read in duration of  $T_{burst\_write}$  :**

$$\text{No of reads} = 2000 \text{ nsec} / 40 \text{ nsec} = 50$$

**5. The depth of FIFO :**

$$\text{Depth} = \text{Burst length} - \text{No of reads} = 100 - 50 = 50$$

## 13.10 Case Study

The FIFO design and case study are described using the following Verilog RTL and the key steps are described in the following template.

```
// FIFO Verilog RTL template

// Module instantiation and port definition

// define the intermediate signals using 'wire' or 'reg'

// Instantiation of FIFO memory buffer

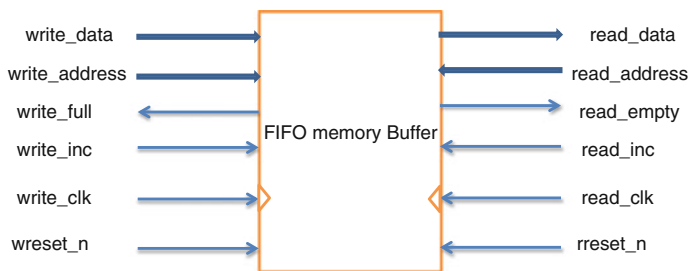
// Instantiation of synchronizers for the write to read clock domain

//Instantiation of synchronizers for the read to write clock domain

// Instantiation of logic for the read empty flag generation

//Instantiation of logic for the write full flag generation
```

FIFO memory buffer top-level pin diagram is shown in Fig. 13.27 and has two



**Fig. 13.27** Pin diagram for FIFO top-level module

different clock domains. Input clock domain or sender clock domain works on the write\_clk and another clock domain, receiver clock domain works on the read\_clk.

```

module FIFO_Memory ( read_data, write_data, write_address,
read_address, write_clk, read_clk, write_en, write_full);

input [data_size-1:0] read_data, write_data;

input [address_size-1:0] write_address, read_address;

input write_en, write_clk, write_full;

parameter data_size =8;
parameter address_size =4;

parameter depth = 1 << address_size;

reg [data_size-1:0] memory [0: depeth-1];

always@(read_address)

assign read_data = memory [read_address];

always@(posedge write_clk)

begin

if (!write_full && write_en)

memory [write_address] <= write_data;

end

endmodule

```

**Example 13.15** Verilog RTL for the FIFO memory buffer

Step1: FIFO memory buffer Verilog RTL (Example 13.15)

Step 2: The Verilog RTL for the write-to-read synchronization logic (Example 13.16)

Step 3: The Verilog RTL for the read-to-write synchronization logic (Example 13.17)

Step 4: The Verilog RTL for the read empty logic (Example 13.18)

Step 5: The Verilog RTL for the write full logic (Example 13.19)

Step 6: The top-level module with module instantiation (Example 13.20)

```
module synchro_write_read ( read_clk, read_pointer_s, rreset_n,
write_pointer);

parameter address_size =4;

input read_clk, rreset_n;

input [address_size:0] write_pointer;

output reg [address_size:0] read_pointer_s;

reg [address_size:0] read_pointer1;

always@(posedge read_clk or negedge rreset_n)
begin
if(!rreset_n)

{read_pointer_s, read_pointer1} <= 0;

else

{read_pointer_s, read_pointer1} <= {read_pointer1, write_pointer};

end

endmodule
```

**Example 13.16** Verilog RTL for the write-to-read synchronizer logic

```
module synchro_read_write ( write_clk, read_pointer, wreset_n,
write_pointer_s);

parameter address_size =4;

input write_clk, wreset_n;

input [address_size:0] read_pointer;

output reg [address_size:0] write_pointer_s;

reg [address_size:0] write_pointer1;

always@(posedge write_clk or negedge wreset_n)

begin

if(!wreset_n)

{write_pointer_s, write_pointer1} <= 0;

else

{write_pointer_s, write_pointer1} <= {read_pointer1, read_pointer};

end

endmodule
```

**Example 13.17** Verilog RTL for the read-to-write synchronizer logic

**Example 13.18** Verilog  
RTL for empty flag logic

```

module read_empty ( read_clk, read_incr, rreset_n, read_empty,
read_pointer, read_pointer_s);

parameter address_size =4;

input read_clk, read_incr, rreset_n;

input [address_size-1:0] read_pointer_s;

output reg [address_size:0] read_pointer;

output reg read_empty;

reg [address_size:0] read_binary;

wire [address_size:0] read_gray_next,read_bin_next;

wire empty_tmp;

always@(posedge read_clk or negedge rreset_n)

begin

if(!rreset_n)

{read_binary, read_pointer} <= 0;

else

{ read_binary, read_pointer } <= {read_bin_next, read_gray_next};

end

always@(*)

begin

read_address = read_binary [address_size-1:0];

read_bin_next = read_binary + ( read_incr && ~read_empty);

read_gray_next = (read_bin_next >>1) ^ (read_bin_next);

empty_tmp = (read_gray_next ==read_pointer_s);

end

always@(posedge read_clk or negedge rreset_n)

begin

if(!rreset_n)

read_empty <= 0;

else

read_empty <= empty_tmp;

end

endmodule

```



**Example 13.19** Verilog  
RTL for full flag logic

```

module write_full ( write_clk, write_incr, wreset_n, write_pointer,
write_pointer_s, write_full);

parameter address_size =4;
input write_clk, write_incr, wreset_n;

input [address_size-1:0] write_pointer_s;

output reg [address_size:0] write_pointer;

output reg write_full;

reg [address_size:0] write_binary;

wire [address_size:0] write_gray_next,write_bin_next;

wire wfull_tmp;

always@(posedge write_clk or negedge wreset_n)

begin

if(!wreset_n)

{write_binary, write_pointer} <= 0;

else

{ write_binary, write_pointer } <= {write_bin_next, write_gray_next};

end

always@(*)

begin

write_address = write_binary [address_size-1:0];

write_bin_next = write_binary + (write_incr && ~write_empty);

write_gray_next = (write_bin_next >>1) ^ (write_bin_next);

wfull_tmp = (write_gray_next ==(!write_pointer_s [address_size: ad-
dress_size-1], write_pointer_s[address_size-2:0]));

end

always@(posedge write_clk or negedge wreset_n)

begin

if(!wreset_n)

write_full <= 0;

else

write_full <= wfull_tmp;

end

endmodule

```

**Example 13.20** FIFO  
top-level module and  
instantiation

```

module FIFO ( write_clk, read_clk, write_incr, read_incr, wreset_n,
rreset_n, write_data, write_full, read_empty);

parameter address_size =4;

parameter data_size =8;

input write_clk, read_clk;

input write_incr, read_incr;

input wreset_n, rreset_n;

input [data_size-1:0] write_data;

output reg [data_size-1:0] read_data;

output read_empty, write_full;
wire [address_size-1:0] read_address, write_address;

wire [address_size:0] write_pointer, read_pointer, write_pointer_s,
read_pointer_s;

//FIFO memory buffer instantiation

FIFO_Memory FIFO ( .write_clk(write_clk),.write_full(write_full),
.write_en(write_incr),.write_data(write_data),
.read_data(read_data),.write_address(write_address),
.read_address(read_address));

//Write to read clock domain synchronizer instantiation

synchro_write_read sync1 ( .read_clk(read_clk),
.read_pointer_s(read_pointer_s), .rreset_n(rreset_n),.
write_pointer(write_pointer));

//Read to write clock domain synchronizer instantiation

module synchro_read_write sync2 ( .write_clk(write_clk),
.read_pointer(read_pointer), .wreset_n(wreset_n),
.write_pointer_s(write_pointer_s));

//Write full logic instantiation

write_full ( .write_clk(write_clk), .write_incr(write_incr),
.wreset_n(wreset_n), .write_pointer(write_pointer),
.write_pointer_s(write_pointer_s), .write_full(write_full));

//read empty logic instantiation

read_empty empty ( .read_clk(read_clk), .read_incr(read_incr),
.rreset_n(rreset_n), .read_empty(read_empty),
.read_pointer(read_pointer), .read_pointer_s(read_pointer_s));

endmodule

```

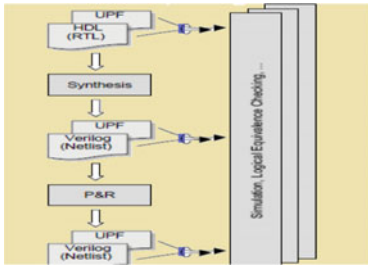
## 13.11 Summary

Following are the key points to summarize this chapter:

1. Clock Domain Crossing (CDC) is critical to fix in the ASIC design and these errors can cause the design failure.
2. For single-bit control signal transfer across the multiple clock boundaries, register the signal at the sending clock domain and avoid the glitches and hazard effect of the combinational logic.
3. Use Multi-Cycle Path (MCP) formulation while passing the single-bit control signals across the clock boundaries.
4. Use the multistage synchronizers in the receiving clock domain while sampling the single-bit control signals at the receiver clock domain.
5. While passing the multiple control or data signals from one of the clock domains to another clock domain, use the consolidated control signal that is one-bit representation of the multiple signals in the sending clock domain.
6. Use the multistage synchronizer in the receiver clock domain while sampling the consolidated control signals.
7. To pass multiple control signals across the clock domains use the MCP formulation.
8. Use the Gray code counters instead of binary counters while passing the data across multiple clock domains.
9. Use FIFO in the data or control path while passing the multiple data bits or control bits.
10. Partition the design using the single clock at the receiving and transmitting (sending) ends.

# Chapter 14

## Low Power Design



Low power is one of the key requirement for ASIC design. The power can be minimized using the different techniques and using the consistent power format. This chapter discusses about the low power design techniques and the need of Unified Power Format (UPF). This chapter also describes about the key UPF commands.

**Abstract** In the modern lower process node ASIC design the power is considered as the major factor. The low power design chips are required in many applications like mobile, computing, processing, and video and audio controller designs. Most of the SOC designs need the low power design support. This chapter discusses about the low power design techniques at the RTL level and the use of the consistent format UPF at the logical design. This chapter is useful for the RTL design engineers to understand the UPF terminology and the key commands for inclusion of the level shifter, retention, and isolation cells. Even this chapter describes about the multiple power domain creation with the UPF commands.

**Keywords** CMOS · Static power · Dynamic power · Switching power · Net power · Parasitic · Multi supply multi voltage · Isolation logic · Isolation cell · Level shifter · Retention cell · Clock gating · Voltage scaling · UPF

### 14.1 Introduction to Low Power Design

In the modern ASIC and SOC designs the power optimization is very crucial. The power requirements for the ASIC or SOC play an important role in the planning of design. The overall power estimations for the chip and the design of low power architecture and microarchitecture are decisive factors in the ASIC design flow. The goal of the chip architect is to design the architecture and microarchitecture for low

power aware designs. As process node has shrunk from 90 to 14 nm in the past decade the voltage levels are dropped substantially.

As power is one of the crucial factors in the design of SOC, it has become the main problem in every category of the design. The power density is measured as watt per square millimeter and it raises with the alarming rate in the SOC designs. So for the SOC design perspective the power or energy management needs to be used in the design from the architecture stage itself. Hence low power design techniques are essential to be used from RTL to GDSII.

Power management is required for all the designs below the process node of 90 nm. As size of the chip has shrunk below 90 nm at the smaller geometry it requires the aggressive management of the leakage current. The primary source of power dissipation in CMOS is leakage current. The leakage current is the summation of the cell leakage current and is state dependent.

The dynamic power is defined as addition of the summation of the internal cell dynamic power and summation of power dissipated due to wires. The following are the few equations which describe the leakage and dynamic power:

$$P_{\text{leakage}} = \sum \text{Cell Leakage}$$

where cell leakage can be computed using the library cell leakage and it is state dependent.

$$P_{\text{dynamic}} = \sum \text{Cell dynamic power} + \sum \frac{1}{2} * C_1 * V * V * T_r$$

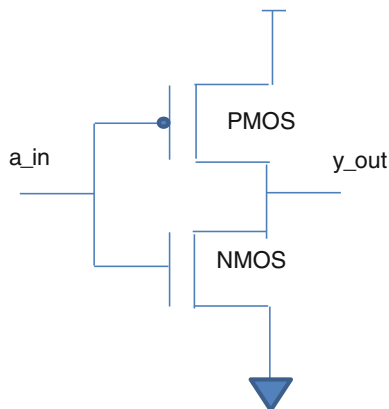
where the  $C_1$  is the capacitive load at pin or net,  $V$  is the voltage level, and  $T_r$  is the toggle rate.

In the past decade, the major interest of designer was to improve the design performance, that is, throughput and latency and frequency and even to reduce the silicon area. But below 90 nm the power management has become the key for the SOC designs. In the present scenario, due to the exponential growth in the field of the wireless and mobile communications and other home electronics intelligent applications the demand is for the complex functionalities and high-speed computations. Even the low power management and the long battery life are key for such kind of applications in the competitive market. It is expected that such kind of devices should be of lightweight, small, cool, and even they should have the long battery life.

## 14.2 Power Dissipation in CMOS Inverter

Consider the ASIC standard cell as inverter shown in Fig. 14.1. As shown in the figure inverter is designed using PMOS and NMOS. PMOS passes strong '1' and NMOS passes strong '0.' At a time either PMOS is on or NMOS is on. But

Fig. 14.1 CMOS inverter



practically the invert cell is represented as equivalent resistance of one of the on transistors, and the equivalent parasitic capacitance is seen at the output port 'y\_out.' Energy stored in the capacitor is dependent on the capacitor value in nano- or picofarad and the voltage applied to the inverter.

The power dissipation for the inverter is computed using the formula  $p = (1/2) * C_s * V^2 * f$ . So the power dissipation for any standard cell is directly proportional to the stray capacitance ( $C_s$ ), applied voltage ( $V$ ), and the frequency. To reduce the overall power for the chip, it is essential to minimize the applied voltage and essential to choose the process technology which can give minimum capacitance at the output and input ports. Due to the high-speed design requirements it is not possible to minimize the speed of the design by reducing the frequency. So there is always trade-off between the power requirements and speed of the design.

Sources of power consumption in CMOS are described in Fig. 14.2.

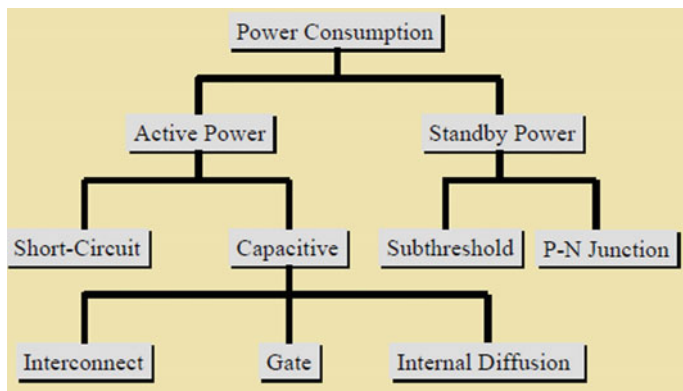


Fig. 14.2 Power consumption sources

So the power dissipation for any CMOS cell is the function of the switching activity, capacitance, voltage, and the structure of transistor. So the power is described as

$$\text{Power} = P_{\text{switching}} + P_{\text{short-circuit}} + P_{\text{leakage}}$$

The total power for any CMOS cell is summation of the dynamic and leakage power.

Dynamic power is summation of the switching power and short-circuit power.

The power dissipation is due to the charging and discharging of internals and net capacitances. The short-circuit power dissipation is due to the gate switching state and it is due to the short circuit between the supply voltage and ground. The following equation describes the switching and short-circuit power:

$$P_{\text{switching}} = \alpha * f * C_{\text{eff}} * V_{\text{dd}} * V_{\text{dd}}$$

where  $\alpha$  is the switching activity,  $f$  is the switching frequency,  $C_{\text{eff}}$  is the effective capacitance, and  $V_{\text{dd}}$  is the supply voltage.

$$P_{\text{short-circuit}} = I_{\text{sc}} * V_{\text{dd}} * f$$

where  $I_{\text{sc}}$  is the short-circuit current during switching,  $f$  is the switching frequency, and  $V_{\text{dd}}$  is the supply voltage.

Dynamic power can be reduced by reducing the switching activity, clock frequency (it reduces the design performance), also using the capacitance and the supply voltage. If faster slew cells are used, then it consumes the less dynamic power and hence cell selection is important in reduction of the dynamic power.

Leakage power is given by the following equation and it is the function of the supply voltage  $V_{\text{dd}}$ , the switching threshold voltage  $V_{\text{th}}$ , and the size of transistor:

$$P_{\text{leakage}} = f \left( V_{\text{dd}}, V_{\text{th}}, \frac{W}{L} \right)$$

In the above equation the  $W$  is the width of transistor and  $L$  is the length of transistor.

Powers saving opportunities at different design stages are listed in Table 14.1.

**Table 14.1** Percentage power saving

Design abstraction stage	%Power saving (%)
System design and architecture	70–80
Behavioral design	40–70
RTL design	25–40
Logic design	15–25
Physical design	10–15

## 14.3 Switching and Leakage Power Reduction Techniques

There are several techniques used to reduce the power and few of the commonly used power management techniques are listed in Table 14.2.

Another few important techniques used in the power management at the various abstraction levels are listed in Table 14.3.

**Table 14.2** Power management techniques

Power management technique	Description
Clock gating and Clock tree optimizations	In this technique the portions of the clock tree which are not used at the instance of time are disabled
Logic restructuring	Use the cone structure to minimize the power. Move the low switching operations back in the logic cone and high switching operations up in the logic cone. This technique is used to reduce the dynamic power at gate-level optimizations
Operand isolations	This technique is effective in reducing the power dissipation in the data path of any blocks by using the enable signals
Logic and transistor resizing	Use the downsizing to reduce the leakage current and use upsizing to reduce the dynamic current by improving the slew times
Pin swapping	Use the swapping gate pins to reduce the power. If the capacitance is lower then the switching can be fast at the gate or pin

**Table 14.3** Efficient power management techniques

Power management technique	Description
Multi- $V_{th}$	Use the multi-threshold libraries for the power reduction. Use the high switching threshold for lesser leakage power but it reduces the design performance. Use the low switching thresholds for the higher performance but it has higher leakage
Multiple supply voltage (MSV islands)	Use the multiple supply voltages for different design blocks
Dynamic voltage scaling (DSV)	In this technique the selected blocks can run at different supply voltages according to the design requirements
Dynamic voltage and frequency scaling (DVFS)	This is used to reduce the dynamic power. In this method the selected blocks of design use different supply voltages and frequencies on fly
Adaptive voltage and frequency scaling (AVFS)	This can be accomplished using analog circuits and in this technique based on the control loop feedback the wide range of voltages are set dynamically
Power gating or power shut-off (PSO)	If the functional blocks are not used then the selected functional blocks are powered off
Splitting memories	If the memories are controlled by software or the data then the portions of memories can be spitted into more number of portions. This is effectively used to save the power by shutting off the portion of memories which are not used



### ***14.3.1 Clock Gating and Clock Tree Optimizations***

This technique is very efficient in reducing the dynamic power. In most of the applications the power is wasted due to unnecessary toggling of the clock signal. If the register clk input is changing, the clock signal toggles on every clock cycle. This is the major reason for the more dynamic power. Even the clock trees are the major sources for the larger dynamic power as they have the larger capacitive load and the switching requires the maximum rate. So if the data is not loaded in the register frequently then significant amount of power is wasted and this can be saved using clock gating technique. The clock gating is at the register level or leaf level and if it is done at the block level, then the entire functional block can be disabled by disabling the clock tree. This reduces the switching and hence reduces the dynamic power.

### ***14.3.2 Operand Isolations***

This technique is effective in reducing the dynamic power dissipation in the data path of any blocks using the enable signals. Most of the times the data path elements are sampled periodically and hence this sampling can be controlled using the enable inputs. During inactive state of enable signal the datapath inputs can be forced to the constant value and hence it reduces the dynamic power due to lesser switching.

### ***14.3.3 Multiple $V_{th}$***

This technique is very effective while optimizing for area, power, and speed using different threshold voltages. Most of the libraries have different switching threshold voltages. The efficient EDA tool used for synthesis can be able to use different library cells of different switching threshold voltages for meeting the area and speed constraints with the lowest power dissipation.

### ***14.3.4 Multiple Supply Voltages (MSV)***

In this technique different functional blocks operate at different voltage levels. As the voltage level reduces, active power is reduced as it is the function of the square of the supply voltage. But it can degrade the design performance. While using this technique it is required to use the level shifter to transfer the signals from one voltage domain to another voltage domain. If level shifters are not used, then the sampling of the signal is an issue.

### ***14.3.5 Dynamic Voltage and Frequency Scaling (DVFS)***

Dynamic Voltage and Frequency Scaling is very efficient technique to reduce the active power consumption. As discussed in the earlier section the power dissipation is proportional to the voltage square, so lowering the voltage has squared effect on the power consumption. In this type of technique, depending on the performance and power requirements, the frequency and voltage can be scaled down on the fly and hence it can reduce the power dissipation. This technique is very effective to optimize the static and dynamic powers due to optimization of the frequency and voltage levels.

### ***14.3.6 Power Gating (Power Shut-Off)***

Power gating or power shut off (PSO) is one of the effective techniques, and in this technique the design modules which are not used are switched off using switches. This is one of the powerful techniques used to reduce the leakage power. In most of the industrial applications the leakage power can be reduced by, more than 90 % using the power gating switches. To design this technique, it needs the clear understanding of the power-down sequence and isolation cells. It is essential to use the isolation logic with the state retention elements and even level shifters to implement the power gating.

### ***14.3.7 Isolation Logic***

This is used at the output of power-down block to prevent unpowered signals and floating signals from power-down block. In the simulation these signals can be denoted by 'X.' Isolation cells are used between the two power domains and connected between the power-off domain and the power-on domain. The reason for isolation cell in the two power domains is to isolate the output of blocks before the power switch off state and need to remain isolated until the power is switched on. In few design scenarios isolation cells can be used to block level to prevent the connection to power-down logic. Consider the block logic as driving power domain and it is in the off state, and then isolation cell can be located in the driving domain to isolate the signals from the driving power domain to the receiving power-on domain.

### 14.3.8 State Retention

During the power-off mode, most of the time it is essential to retain the state of the registers. The state of the registers is useful during the power recovery. In most of the low power designs the state retention power gating flip-flops are used and these flip-flops are called as SRPG. Most of the EDA tool cell libraries have the SRPG cell.

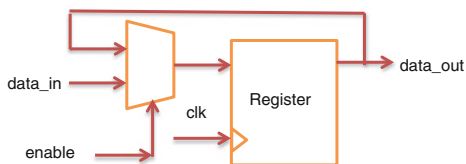
## 14.4 Low Power Design Techniques at the RTL Level

In the present scenario, there are many low power design techniques at the RTL and gate level. It is essential for the design team to understand about the low power goals to define the techniques uniformly by ensuring the consistency and predictability in the overall design cycle. Most of the SOC designs use the low power design techniques using power analysis and optimization issues. This section focuses on the low power design technique.

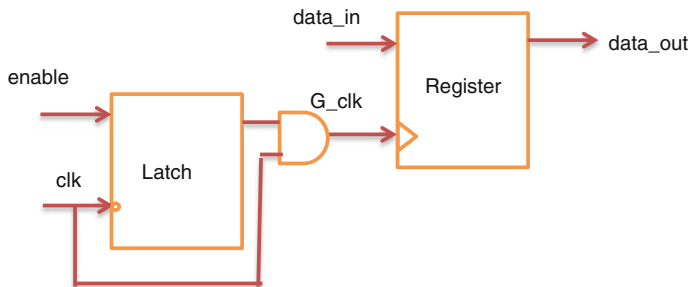
1. Modeling and power estimation: For the low power design and the management of power for any SOC it is essential to prepare the library models with the required power data. It is required to develop the transistor-level models for the custom blocks. The common practice in the SOC design at the RTL level is the use of power compiler to understand the power consumption based on the switching activity information from the RTL simulation data. This technique is useful for estimation of the power consumption at early stage of the design. Another important point to be considered at the gate level is to develop the glitch-free low power designs and state and path dependencies' support. As gate-level analysis is more accurate as compared to the RTL level analysis it is essential to use the time-based analysis based on the peak power and hot spots.
2. Clock Gating: Use the clock gating technique using the clock gating cells to minimize the power during the RTL design. Clock gating can be implemented by identifying the synchronous load enable register banks. Clock gating can be implemented using the gating of clock with the enables instead of recirculating of the data when enable is inactive. If power compiler is used at the RTL level then it automatically optimizes the static, dynamic power dissipation with the delay and area to meet the design constraints.

Clock gating stops the clock and forces the original circuit in the zone of no transition. In the practical scenario if we consider the functional block as

```
always@(posedge clk)
begin
if(enable)
data_out<=data_in;
end
```



**Fig. 14.3** Logic without clock gating



**Fig. 14.4** Logic with clock gating

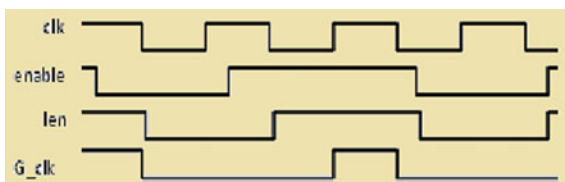
The above block can generate the synthesis result as shown in the Fig. 14.3.

The above-generated logic is without clock gating and has the higher power dissipation. To reduce the power consumption the clock gating logic needs to be used and can be designed by eliminating the multiplexers at the input thus avoiding the recirculation of data. This results in the area and power savings and reduces the power consumption in the clock network. The synthesized logic using clock gating is shown in Fig. 14.4. The timing sequence is shown in Fig. 14.5.

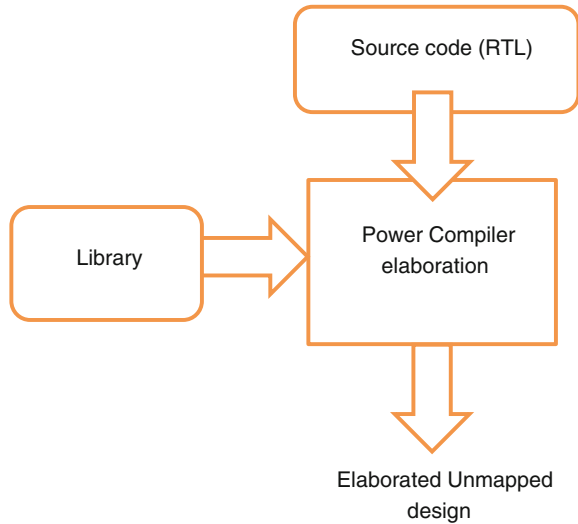
The use of clock gating has a drawback that the logic used to implement the clock gating technique is redundant and hence there can be issues in the testing and verification. Another important point needs to be kept in mind is that it is essential to stop the glitches and hazards on enable signal and this is achieved using the transparent latch between the enable and the AND logic gate.

Clock gating can be efficiently implemented using the power compiler from synopsys. Use the command **set\_clock\_gating\_signals**. Figure 14.6 illustrates the inputs and outputs used for the power compiler.

**Fig. 14.5** Timing sequence for the clock gating



**Fig. 14.6** Power compiler inputs and outputs



The outcome of the power compiler is the elaborated unmapped design. Power compiler uses the inputs as source RTL code and library to optimize for the low power.

The following are few of the key points need to be considered while implementing the clock gating using the power compiler.

1. General clock gating can be included or excluded from the design for the hierarchical modules. The command used is **set\_clock\_gating\_signals**. The care needs to be taken by the designer while using the power compiler for the same. Each design should have the single command line for both the inclusion and exclusion of the clock gating.
2. If the design has multiple registers and few of the registers need to be excluded from the clock gating strategy then they should have the separate enable signal. If same enable signal is used then it generates the same clock gating for the entire register bank. For example, if the data bus is defined as `data_in[7:0]` with the registered inputs and if the lower nibble '`data_in[3:0]`' needs to be excluded from clock gating, then it should have a different enable and `data_in[7:4]` should have a different enable.
3. Clock gating signals as single bit or multiple bits have added advantage as it avoids the recirculation of the data by removing the multiplexers. But it can consume more area and additional power due to the clock gating logic.
4. Do not use clock gating for the master-slave flip-flops. Generally, it is common practice that clock gating logic is used at the slave flip-flop if the clock gating conditions are met. Such design may not perform the desired operation. Use the command `set_clock_gating_exclude` to exclude the master-slave flip-flops.

5. While using the clock gating it is common practice to use the minimum bus width. The minimum bus width can be of 5 or more. Use the command `set_clock_gating_style_minimum_bitwidth`.
6. In most of the design practices at the RTL level if the procedural ‘always’ blocks are used and if it consists of ‘case’ with the ‘default’ clause or conditional expressions like ‘if-else’ then modify the RTL by including the default condition in every ‘if-else’ statement. Example 14.1 describes the modification of the procedural block using ‘default’ as ‘else’ clause.
7. If same enable is shared by the multiple register banks, then the power compiler feature can be used to share the clock and enable signal to multiple register bank. This is used to save the overall area. Consider Example 14.2 and it has two different procedural blocks, and then the same clock gating logic can be used for both of the procedural blocks.
8. Use the simple clocking strategies for the automatic clock gating insertion. If the number of clock domains is minimum then it gives simplified timing analysis and clock tree synthesis. The lower down modules can have enable signals instead of dividing the clock. Use the `set-don't_touch_network` command to avoid the compilation changes on the clock network. During the multiple step compilation process this avoids the changes on the clock gating logic.

**Example 14.1** Rearranging of RTL for the power saving

```

case(a_in)
  2'b00: if (b1_in) c_in =d1_in;
  2'b01: if (b2_in) c_in =d2_in;
  default : c_in = e1_in;
endcase

The above Verilog RTL can be modified as

case(a_in)
  2'b00: begin
            if (b1_in) c_in =d1_in;
            else c_in=e1_in;
          end
  2'b01:begin
            if (b2_in) c_in =d2_in;
            else c_in =e1_in;
          end
endcase

```

**Example 14.2** Common clock enable for multiple procedural blocks

```

always @ ( posedge clk or negedge reset_n)
begin : block_1
    if (~reset_n)
        data_out <= 1'b0;
    else if (enable)
        data_out<=data_in;
end

always @ ( posedge clk or negedge reset_n)
begin : block_2
    if (~reset_n)
        data_out_1<= 1'b0;
    else if (enable)
        data_out_1<=data_in_1;
end

```

9. Use the simple set and reset strategies. Complex set and reset strategies may result in the design logic which is prone to issues at the gate-level functional debugging. The care needs to be taken by the designer to have the proper logic partition for synthesis while using the internal set and reset signals.
10. Clock balancing and the clock buffer signal insertion need to be used efficiently to have efficient clock tree synthesis (CTS). CTS tools work by adding or moving the buffers, resizing of cells along the clock tree network to manage the required skew and the insertion delay.

## 14.5 Low Power Design Architecture and UPF Case Study

Unified power format (UPF) is the standard used to design electronic systems by considering the power as the feature. The standard is used for low power ASIC designs. The reasons for using UPF are as follows:

1. There is no method which can support accurate management and distribution of low power at the HDL level abstraction.
2. Vendor-specific power formats are inconsistent and are prone to bugs due to inconsistent specifications.

3. UPF provides the following and can be used consistently in low power ASIC designs
  - a. Power distribution architecture
    - i. Define the power domains
    - ii. Define power switches
    - iii. Define power rails
  - b. Power strategy
    - i. Creation of power state tables
  - c. Set and map
    - i. Isolation
    - ii. Retention
    - iii. Level shifter
    - iv. Switches

UPF is IEEE 1801 standard and can be used throughout the design flow for power aware design intent. Example 14.3 describes the use of UPF at various stages.

### 14.5.1 Isolation Cells

As discussed already the isolation cells are used at the output of power-down block. The isolation cell can be set using the following UPF command. Figure 14.7 shows the design using isolation cell.

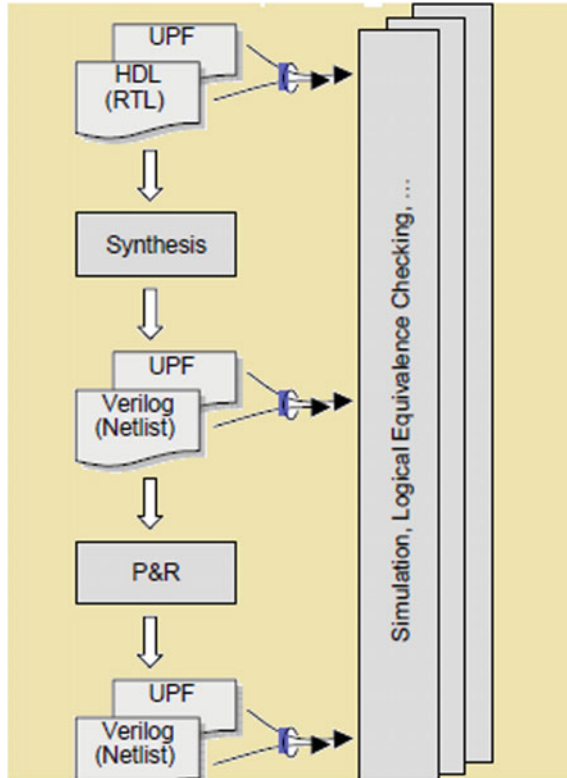
#### Set isolation cell

```
set_isolation iso3
-domain PDgreen
-isolation_power_net Vbu
-clamp_value 0
-applies_to outputs
```

#### Set isolation control

```
set_isolation_control
iso3
-domain PDgreen
-isolation_signal CPU_iso
-location self
```





Example 14.3 UPF at various design stages [1]

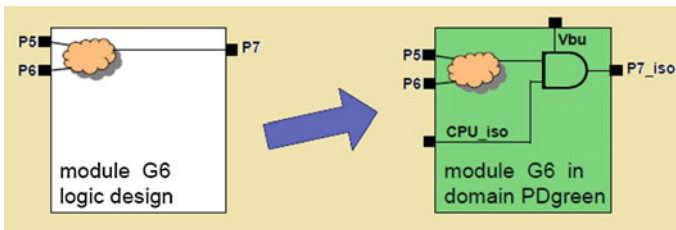


Fig. 14.7 Setting of isolation cell in logic design

### 14.5.2 Retention Cells

As discussed already in the above section the retention cells are used to retain the state of key registers during power-off state. Figure 14.8 shows the setting of the retention cell in the design.

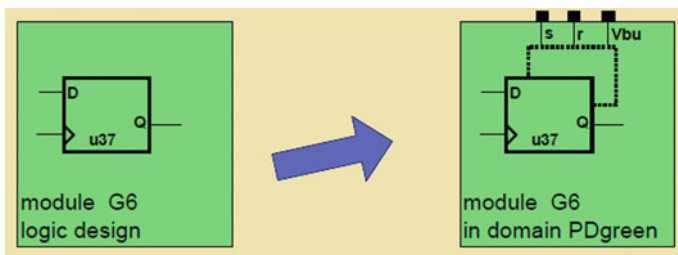


Fig. 14.8 Setting of retention cell in logic design

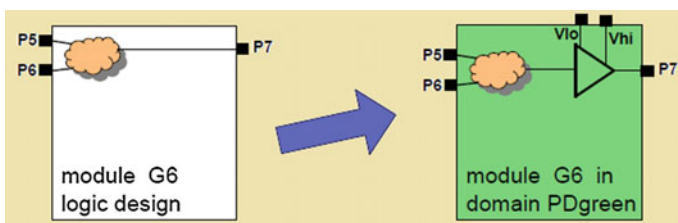


Fig. 14.9 Setting and mapping of level shifter in design

**Set retention cell**

```

set_retention ret3
    -domain PDgreen
    -retention_power_net Vbu
    -elements { u37 }
    
```

**Set retention control**

```

set_retention_control
    ret3
    -domain PDgreen
    -save_signal s
    -restore_signal r
    
```

### 14.5.3 Level Shifters

Level shifters are used to translate from one voltage level to another voltage level. The translation can be from low to high voltage level or high to low voltage level. Set and map level shifter can be achieved by using the following UPF commands. Figure 14.9 shows the use of command to set and map the level shifter.

The key points to consider for the same are

1. Pick the correct power domain
2. Select input or output ports or both
3. Use UP-SHIFT or Down-shift rule
4. Define the location

#### Set level shifter

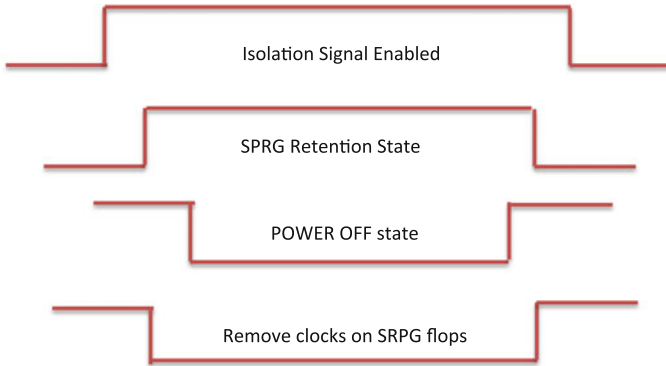
```
set_level_shifter my_ls
    -domain PDgreen
    -rule low_to_high
    -location self
    -applies_to outputs
```

#### Map level shifter

```
map_level_shifter_cell
    ls_L2H
    -domain PDgreen
-lib_cells { /lib/ls_123 }
```

### 14.5.4 Power Sequencing and Scheduling

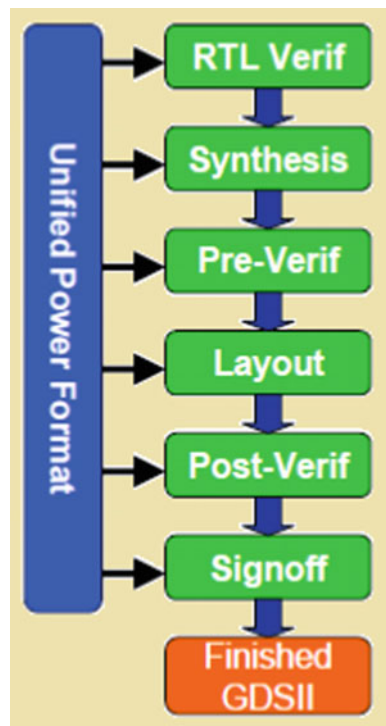
Specific sequence is generally followed for the power down. The sequence includes isolation, state retention, and the power shut-off. For the power-up cycle the opposite sequence needs to be followed. During power-up cycle it is recommended to have the specific reset sequence. Following timing sequence gives information about the power-up/down sequence.



For the multiple clock domains with different power sequences and the multiple clock gating with few common power control signal, it requires the higher verification efforts to ensure the correct sequencing for the power-on and power-off.

The UPF can be used from the RTL to GDSII and the basic UPF support is shown in Fig. 14.10. During the verification using the UPF, the functional and

**Fig. 14.10** UPF from RTL design to GDSII [2]



power intent should be analyzed and need the robust verification using the advanced verification techniques.

#### 14.5.4.1 Creation of Power Domains

The power domains can be created using the following UPF command:

```
create_power_domain domain_name
    [-elements list]
    [-include_scope]
    [-scope instance_name]
```

For example creating the power domain having name pdA, the UPF command used is given below and the outcome is shown in Fig. 14.11:

```
create_power_domain pdA -include_scope A
```

#### 14.5.4.2 Create Supply Port

The supply port can be created using the following UPF command:

```
create_supply_port port_name
    -domain domain_name
    [-direction <in | out>]
```

For example creating the supply port with the name spAOn, the command used is given below and the outcome is shown in Fig. 14.12:

```
create_supply_port spAOn - domain pdA
```



Fig. 14.11 Creation of power domain [2]



Fig. 14.12 Creation of supply port [2]

### 14.5.4.3 Create Supply Net

The supply net can be created using the following UPF command:

```
create_supply_net net_name
  -domain domain_name
  [-reuse]
  [-resolve < unresolved
    | one_hot
    | parallel >]
```

For example creating supply net named as RET, the UPF command used is given below and the outcome is shown in Fig. 14.13:

```
create_supply_net RET -domain pdA
```

For example creating supply net named as PR, the UPF command is given below and the outcome is shown in Fig. 14.14:

```
create_supply_net PR -domain pdA
```

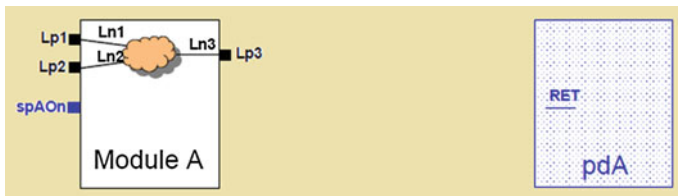


Fig. 14.13 Creation of supply net RET [2]

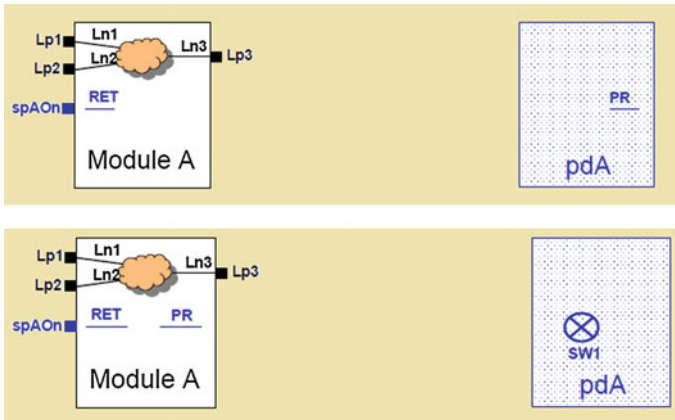


Fig. 14.14 Creation of supply net PR [2]

#### 14.5.4.4 Create Power Switch

The power switch can be created using the following UPF command:

```
create_power_switch switch_name
    -domain domain_name
    -output_supply_port { port_name supply_net_name }
    {-input_supply_port { port_name supply_net_name }}*
    {-control_port { port_name net_name }}*
    {-on_state {state_name input_supply_port
        {boolean_function}}}*
    [-on_partial_state { state_name input_supply_port {
        boolean_function }}]*
    [-ack_port { port_name net_name [{boolean_function}] }]*
    [-ack_delay { port_name delay}]*
    [-off_state { state_name {boolean_function} }]*
    [-error_state { state_name {boolean_function} }]*
```

For example creating the power switch SW1, the UPF command used is given below with the net outcome shown in Fig. 14.15:

```
create_power_switch SW1 -domain pdA
    -input_supply_port {inp RET}
    -output_supply_port {outp PR}
```



Fig. 14.15 Power switch creation [2]

### 14.5.4.5 Connect Supply Net

The connection for supply net can be created using the following UPF command:

```
connect_supply_net net_name
    [-ports list]
    [-pins list]
    [< -cells list |
    -domain domain_name >]
    [< -rail_connection rail_type |
    -pg_type pg_type >]*
    [-vct vct_name]
```

For example connecting the power supply net, the command used is given below and the net outcome is shown in Fig. 14.16:

```
connect_supply_net RET -ports {spAOn}

set_domain_supply_net pdA
    -primary_power_net PR
    -primary_ground_net VSS
```



Fig. 14.16 Connecting the supply net [2]



## 14.6 Summary

The following are the key highlights to summarize this chapter:

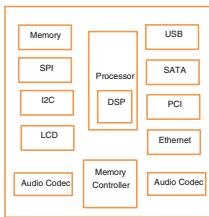
1. The dynamic power is defined as addition of the summation of the internal cell dynamic power and summation of power dissipated due to wires.
2. Dynamic power can be reduced by reducing the switching activity, clock frequency (it reduces the design performance), also using the capacitance and the supply voltage.
3. Operand isolation is effective in reducing the dynamic power dissipation in the data path of any blocks using the enable signals.
4. Dynamic voltage and frequency scaling are very efficient technique to reduce the active power consumption.
5. The retention cells are used to retain the state of key registers during power-off state.
6. Level shifters are used to translate from one voltage level to another voltage level. The translation can be from low to high voltage level or high to low voltage level.
7. Unified power format (UPF) is the standard used to design electronic systems by considering the power as the feature. The standard is used for low power ASIC designs.
8. This is used at the output of power-down block to prevent unpowered signals, floating signals from power-down block.
9. Power gating or power shut-off (PSO) is one of the effective techniques, and in this technique the design modules which are not used are switched off using switches. This is one of the powerful techniques used to reduce the leakage power.

## References

1. IEEE1801 low power design standard. [www.ieee.org](http://www.ieee.org)
2. Power Compiler Reference Manual: Synopsys Inc. [www.synopsys.com](http://www.synopsys.com)

# Chapter 15

## System on Chip (SOC) Design



The System on Chip can be realized and prototyped by using FPGAs. The SOC consists of many complex blocks like processors, arbiters, memories, peripherals. These blocks are discussed in this chapter. This chapter even focuses on the generalized SOC architecture and even the SOC design flow. Finally the case study is included for the better understanding.

**Abstract** SOC's are complex density ASICs and need to be validated using the FPGAs. In the present scenario there is more demand for the FPGA prototyping to realize the ASICs. Single or multiple FPGA can be used to prototype the desired SOC functionality. This chapter focuses on the discussion on the SOC components, challenges, and the SOC design flow. Even the individual key SOC block coding is discussed in this chapter.

**Keywords** SOC · IP · Timing accurate model · Cycle accurate model · Data hazards · Structural hazards · Pipelining · STA · Timing closure · FPGA prototyping · ASIC porting · Verification plan · Test plan · DFT · UPF · ASIC porting · Control path · Data path · Control hazard · Arbitration · IO · Timers · Counters · Memories · Microcontroller · Microprocessor

## 15.1 What is System on Chip (SOC)?

System On Chip (SOC) is designed by using ASIC design flow and for proof of concepts PLDs are used. In the present scenario the designs are complex in nature and consist of multiple functional blocks to perform the desired operations with high design performance. The main important SOC design challenge is to have lower power, high performance, and less die area.

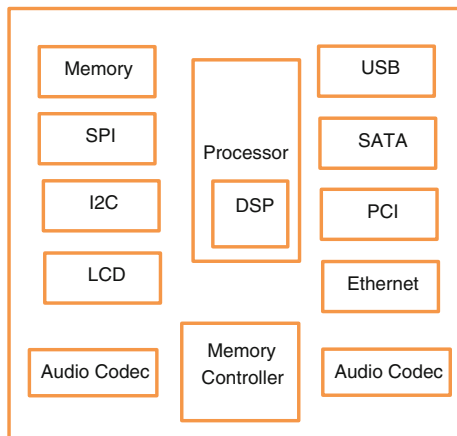
As SOC complexity has increased over the past decade it has become extremely important to detect the defects in the SOC's during early stage of design cycle. The best and affordable way is to use the modern FPGAs to realize or prototype the design. In the present scenario most of the complex designs are prototyped by using modern FPGA architectures.

It is essential to understand about, why the FPGA prototyping has become popular during this decade? The main reason is the lesser non recurring investment and the availability of the high performance computing and reprogrammable block in the FPGA. SOC's consists of processor, IO interfaces such as Ethernet, SATA, USB, UART, SPI, I2C, high performance DSP computational capabilities, video and audio codecs, and high speed memory controllers like DDR II or DDR III. Modern FPGAs are used for SOC prototyping as they have most of the capabilities listed above to achieve the high performance.

## 15.2 SOC Architecture

In the present decade IP and SOC complexity has increased so much. There is demand for SOC design and FPGAs with the high density functional blocks used for validation of SOC. This is also called as ASIC or SOC prototyping. we consider that typical SOC has processor core, various memories, and clock source as PLL, multiple power domain functional blocks, peripherals, communication interfaces, and analog-to-digital and digital-to-analog converters. The important point in the design of SOC is to partition of the hardware and software blocks. In the present scenario the FPGAs are used for SOC prototyping due to reconfigurable capabilities and to accelerate the performance of design due to use of soft and hard IPs in it.

The different blocks for SOC are shown in Fig. 15.1. If we consider any complex SOC then it consists of the different communication interfaces such as USBs, Bluetooth and most of the SOC's support the standard protocols. For any SOC design it is essential to achieve the area, speed, and power constraints. Achieving the required design functionality with the constraints is one of the challenges due to the availability of lesser time to design and market the product due to high demand of new features and functional requirements. SOC design always needs the realistic plan, resources, and availability of necessary validation testing and prototyping setup.

**Fig. 15.1** SOC design blocks

## 15.3 SOC Design Flow

The SOC design flow for the design and validation is shown in Fig. 15.2. As shown in figure it has multiple steps which includes the design feasibility and implementation, FPGA prototyping and testing and ASIC porting. The key steps are discussed in the subsequent section (Fig. 15.3).

### 15.3.1 IP Design and Reuse

Most of the SOC uses intellectual properties (IPs). But as designer, it is important to validate the IPs in SOC's due to available features, timing requirements, and functionality. The important parameter in IP design is the overall functionality of the design. The IPs are getting sold in the semiconductor market due to its features, timing performance, and low power requirements. If we consider simple tablet then the tablet selling point in the market is due to the availability of the functional features. The IPs are not only sold in the market due to the only interfaces.

Most of the SOC design team always uses the third party functionally and timing proven IPs. Instead of spending the time on design of IPs most of the SOC design team uses multiple IPs required according to the desired functional requirements. All the required IPs can be integrated together according to the speed and power requirements. Although there is challenge in overall integration of IPs and that challenge can be overcome by understanding the architecture details of IPs, timing, and power figures of IPs. The IP can be soft IP or hard IP. The IP vendor companies can provide the synthesizable and process independent RTL, or netlist with the necessary timing information with high performance user friendly interfaces.

The IPs should exhibit the required functionality and should be delivered with the synthesizable RTL, synthesis scripts, design constraints, and interface details.

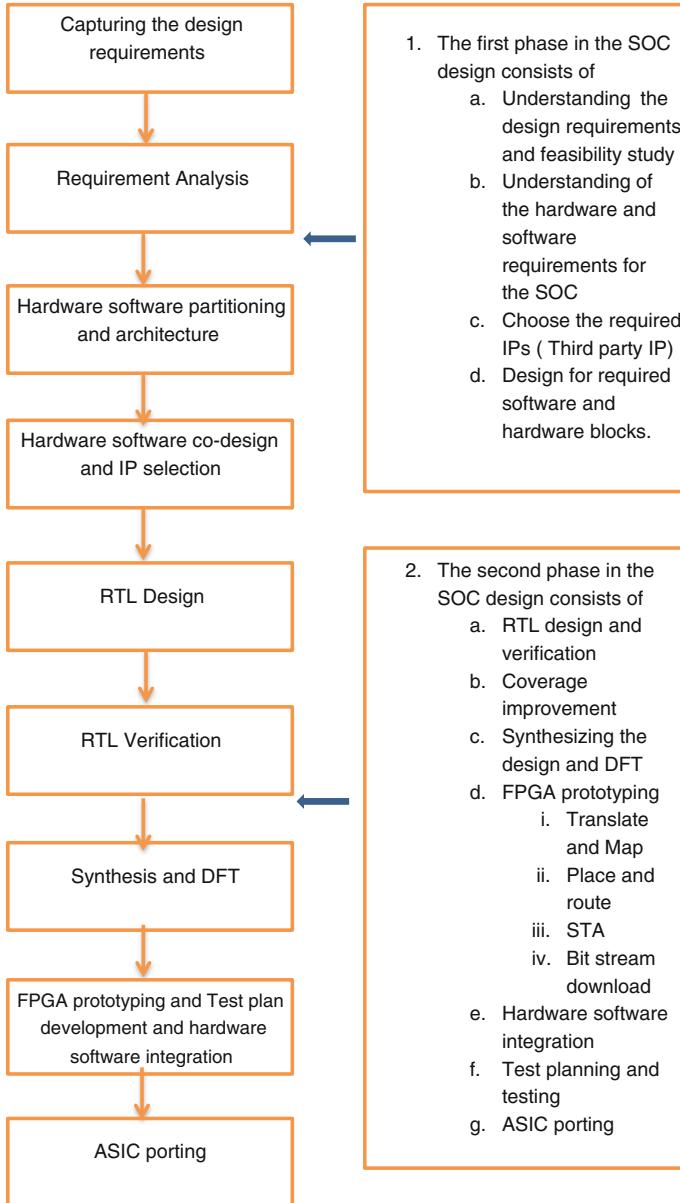
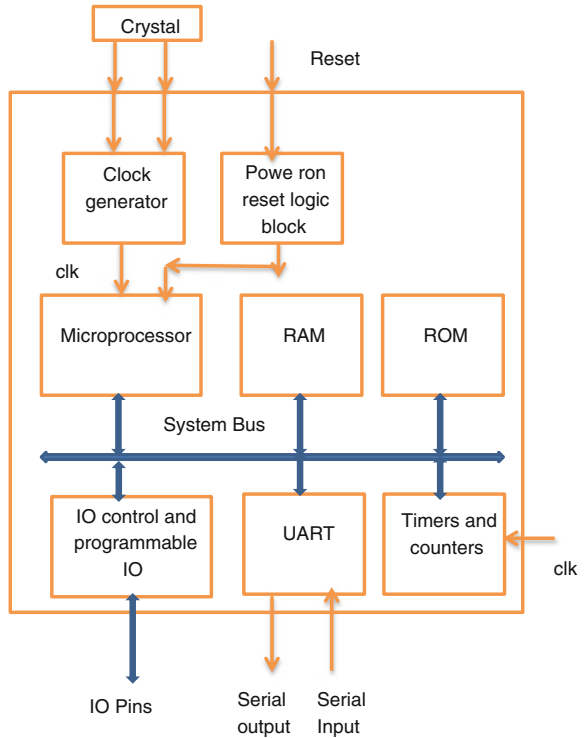


Fig. 15.2 SOC design flow

Then it becomes easy for IP integration and validation of SOC in lesser time. The reason for growing complexity of SOC is due to following few factors:

**Fig. 15.3** Generalized SOC architecture



1. Requirement of number of features with limitation on area that is size of SOC.
2. Less time to market.
3. High speed requirements.
4. The multiple power domain designs and requirements.
5. Multiple clock domain and clock tree structure.

The complex SOCs have always challenges in the placement and routing and meeting the power requirements for the design. Even timing analysis and meeting the timing requirement is one of the challenges.

### 15.3.2 SOC Design Considerations

Capturing the design requirements and analysis of the design is the first important step in the SOC design. The Input for this phase is the design or product specifications provided by the client or end user. The analysis involves the feasibility study of all the features provided. The feasibility study is an important phase as during this phase it can be easily understood about the risks in the implementations as well as dependability during implementation. The feasibility study is needed to be done for all the features by keeping in mind the time to market. This study gives

roadmap and challenges involved during SOC design implementation cycle and useful for even creating various versions with the milestone details. The design specifications are analyzed and understood during this phase.

### ***15.3.3 Hardware Software Codesign***

This is also called as design partitioning; the design has to be partitioned into hardware and software. The important point of consideration is while partitioning the design; how parallel execution needs to be incorporated in the design? In the present scenario as SOC's are complex the functionality can be implemented using the parallelism in the design which in turn can improve the design performance. The complex computational task or algorithms need to be partitioned during the design partitioning phase. Most of the complex computational blocks need to be implemented using hardware. Design partitioning is important and decisive phase to define what need to be implemented using software? And what need to be implemented using hardware?

For example consider the design of video decoders which needs multiple frame support. The video decoder can be efficiently implemented using hardware and even the parallelism can be incorporated for the few decoder features. The high computational DSP functional blocks which need filters like FFT, FIR, and IIR or high speed multipliers can be effectively and efficiently implemented using hardware.

Let us consider the scenario of protocol implementation, most of the protocol like Ethernet, USB, and AHB can be efficiently implemented using hardware–software codesign. These algorithms should be functional and timing proven This can have advantage to overcome and to reduce latency in the design. For most of the protocol implementation it is essential to consider software and hardware design partitioning and design cycle with the overall overheads on the design.

The major challenge in the hardware–software design partitioning is the analysis of throughput and power requirements. For example consider the scenario in SOC design where fixed length packets need to be transferred over the fixed time interval. If the design is implemented by using hardware then care needs to be taken such that there should be minimum interaction between the hardware and software. To minimize the interaction between hardware and software, the strategy can be used by using FIFO buffers and timers.

### ***15.3.4 Interface Timings***

FIFO is first in first out memory buffer and can be used to hold the packet information depending on the depth of FIFO. At the start of data transfer FIFO can interact with procedural calls defined in the software. To track the time duration the timers can be implemented using hardware which can have communication

interface with FIFO to indicate the end of timing intervals. Such type of hand-shaking mechanism can be implemented easily using hardware.

The design architecture for both the hardware and software activities can be created by considering power aware design and throughput requirements.

#### **15.3.4.1 Interface Details and Timing Requirements**

For every SOC, it is essential to have the functional and timing proven bus interfaces. In most of the applications Advanced High Speed Bus protocols are used. These protocols need to be validated for the functional and timing correctness of the design. IO interfaces need to be targeted for the high speed data transfer. There are many different kinds of IO interfaces used in SOC designs. These IOs can be general purpose, differential IOs and high speed IOs.

#### **15.3.4.2 Reset Clock Requirements**

Clock distribution network is used to provide the uniform clock skew to all the registers in the SOCs. The clocking policy plays the crucial role in overall design performance. The uniform clock skew can be achieved by using the suitable clock tree by using clock tree synthesis. Use of single clock structure or multiple clock domain structure need to be decided at the architecture level. Also, the uses of synchronous or asynchronous logic need to be defined at the architecture level. Reset can be asynchronous or synchronous and needs to be defined at the architecture phase of SOC.

### ***15.3.5 EDA Tool and License Requirements***

Choose the required necessary EDA tools and licenses for FPGA prototyping of a SOC and for ASIC porting. The most industry standard tools are

Simulator: Questasim and VCS

Synthesis: Synpilfy pro and Synopsys DC

STA: prime time (Synopsys PT)

### ***15.3.6 Developing the Required Prototyping Platform***

For SOC validation use the necessary prototyping and development platform. Prototyping platform consists of use of multiple FPGA boards to realize and validate SOCs, IP required, DSP functionality required, memories, and general



purpose processors required. The availability of desired prototyping boards with the necessary interfaces to realize SOC and use debug or testing setup.

Most of the SOCs are tested by using the test setup consisting of available EDA tools and logic analyzers. At the start of the SOC design cycle, architect analyses the design and functional requirements and according to the requirement of speed and estimation of gate count the prototyping platform can be build. Here the overall important factors are time to market, budget allocation, and design time requirements. If DSP capabilities are available in FPGA then it is wise to implement the DSP functionality on FPGAs.

### ***15.3.7 Developing the Test Plan***

For complex gate count SOCs, the necessary test cases need to be developed with the required test vectors. The features can be extracted using top level functional specifications and the required test cases can be documented in the test plan document. The test vectors developed can have significant impact on the quality of the verification to achieve the coverage goals. The test cases can be documented as basic, corner, and the random test cases. The constrained random verification with the required coverage goals can be achieved by using the required necessary test cases and test plan.

### ***15.3.8 Developing the Verification Environment***

Use the verification languages like Verilog and high level verification languages like System Verilog or System C; for early detection of bugs and to achieve the coverage goals. The verification planning to improve the overall design quality by capturing the bugs during early design cycle is always crucial in the large gate count SOC designs. The overall objective is to achieve the required and designed functionality in less time. The verification environment needs to be build to achieve the coverage goals. The verification architecture can have the necessary bus functional models and the drivers, monitors, and scoreboards for robust checking of the design functionality. The overall verification planning and creation of environment is with goal to achieve the automation to minimize the time requirement to complete the functional checks in the lesser amount of time duration.

### ***15.3.9 Prototyping Using FPGAs***

At the architecture and microarchitecture level the gate count estimation is done for the SOCs. As discussed already the prototyping development can have multiple

FPGAs with the required high speed interfaces. Depending on the complexity of design FPGAs can be chosen. The main criteria are use of FPGA for the lesser power and more speed. The following are key important points need to be considered while prototyping using FPGAs:

1. Use of FPGA functional blocks to meet the required area requirements. Choose the suitable FPGA platform and use the 70 % of FPGA resources.
2. The area, speed, and power constraints need to be extracted at the chip level and at the block level.
3. Use the block level constraints while synthesizing the blocks and use the chip level constraints at the top level.
4. If high performance DSP algorithms need to be coded then use the DSP functional macro blocks to realize the high computational DSP filtering and the processing algorithms.
5. Try to choose the FPGA platform with required high speed interfaces such as USB, Ethernet, PCI, and memory controllers.
6. Choose the mechanism to interact between software and hardware.
7. Choose the required tool options for auto place and route of design to meet the design constraints.
8. FPGAs should have the capability to achieve the functionality at higher speed.

Low power requirements for the design: Most of the FPGA demands low power in the today's market scenario. SOCs can be designed to meet the desired power. Use the Unified Power Formats (UPFs) to achieve the desired low power requirements.

### ***15.3.10 ASIC Porting***

After performing the realization and validation of SOC using FPGA ,the design needs to be migrated to an ASIC. For quick realization of ASIC, designer need to do following:

1. Replace the clock gating logic with the equivalent component from the ASIC library.
2. Insert DFT and check for the stuck at fault coverage.
3. Use the low power intent design using UPF.
4. Use the block level and chip level constraints while migrating from FPGA to ASIC flow.
5. Synthesize the design for the required constraints.
6. Implement the physical design using the design flow for the required area, speed, and power.

## 15.4 SOC Design Challenges

While designing SOC there might be many design challenges and few of them are listed below.

1. *Use of the modeling abstraction levels* In the practical scenario different modeling levels are used from the design specifications to fabrication of chip. It is good decision to use the different level of abstractions while design of SOC.
  - a. *Functional modeling* To describe the functionality and to get the valid and accurate output by using the simulators.
  - b. *Cycle accurate modeling* To understand the required number of cycles consumed while performing the operation.
  - c. *Event level modeling* To understand the number of events within a clock cycles are accurate or not?
  - d. *Memory accurate modeling* To understand the memory contents and layout is accurate or not?
  - e. *Transaction level modeling* To understand for the number of transactions are accurate or not?
2. *RTL design* Efficient RTL design description and synthesizable RTL is one of the key challenge and ASIC SOC design engineer needs to take care of following:
  - a. Order of continuous assignments and loop free design. The outcome is latch free synthesis results.
  - b. Defining hierarchy of design and having efficient design partition.
  - c. Registering inputs and outputs for the module.
  - d. Uses of each register assignment in single clock domain.
  - e. DFT friendly RTL design and low power aware RTL.
  - f. Properly use blocking and nonblocking assignments.
3. *RTL verification* The goal is to detect the bugs during early design cycle and to achieve the coverage. So, the main challenge is to understand the usage of event-driven or cycle accurate simulators and use of their features. While creating the testbench architecture, care need to be taken for the self checking testbench and design test automation for the higher coverage. Use of the transport and inertial delays during the verification and using zero delay models is another key challenge for the robust verification.
4. *Synthesis* The goal should be to meet the desired power, speed, and area requirements. For low power designs, use the isolation cells, retention cells, level shifters, and clock gating logic. For speed improvement, use the techniques like register balancing, pipelining, and register retiming. For area minimization, use the techniques like multiplex decoding, grouping, and constant data propagation.
5. *Hazard free designs* For any efficient ASIC SOC design, it is recommended not to have the hazards. There are potential issues in the design due to hazards for

example write after write hazard can create the potential issues in the design if second write does not happen properly after first write of the data. Following are the few important points need to be considered for the hazard free design.

- a. *Data hazard* Can be potential problem if the data or address is not computed or arrived at the required time stamp.
  - b. *Structural hazard* Can be potential problem due to the limited number of resources to perform the multiple activities at a time. To overcome these hazards use the registers and sequence the operations using the pipelined structure. Following are few examples for the structural hazards:
    - i. Memories with the limited number of ports and less latency.
    - ii. Non-pipelined designs and limited number of processing units.
    - iii. Implementation of multiplier algorithms without the pipelining or booth multiplication.
  - c. *Control hazards* Can be potential problem due to late arrival of control signal or it is not clear when to perform the operation?
  - d. *Read and write hazard* Can be potential problem if the read and write operations are performed during the same time stamp.
  - e. *Timing estimation and analysis* The challenge is to meet the required timing for the SOC and challenges are following:
    - i. Use the pipelined design with the required pipelined stages.
    - ii. Use the grouping technique and logic duplications for the clean register to register paths.
    - iii. Use the techniques to reduce the critical path timing delays.
6. *Interface and protocol implementations* Most of the SOC design use the protocol and as discussed earlier meeting of the timing performance at the interface level is also an important aspect for the efficient SOC. Following can be few points need to be considered while modeling the protocols and interfaces.
- a. Use of the handshaking mechanism for the transaction notification.
  - b. Use of the general purpose IOs and the special IOs for the interfaces.
  - c. Understanding the timing details at the pin and signal level.
  - d. Use of serializer, deserializer and parallelism while modeling the protocols.
7. *SOC components* Selecting the required SOC components or describing the SOC RTL design is one of the key challenges. The main SOC components can be microprocessors or microcontrollers, IOs, arbiters, memories, general purpose controllers, interrupt, and DMA controller. Describing the RTL for each and every individual component is one of the key challenges as goal is to achieve the required area, speed, and power.
8. *Design Implementation and Testing* After completion of the hardware and software component design, the integration of hardware and software is the major challenge due to the interface synchronization requirements. The testing of the SOC needs the efficient verification and testing plan to test the features covered.

## 15.5 Case Study

The SOC design case study for the moderate complex design is discussed in this section. As discussed in the above section, SOC consists of the microprocessor or microcontroller to perform the processing operation on the multiple operands, the memory banks RAM and ROM, general purpose IO and control mechanism, counters and timers, and UART. For easy understanding of SOC, the complex modules like DSP controllers, DMA controllers, video controllers, and complex arbiter are not discussed in the case study. Readers are encouraged to use the fundamental design concepts to describe the architectures and to code the RTL for the above complex modules.

## 15.6 SOC Design Blocks

The key SOC design blocks and the Verilog RTL for these blocks are discussed in this section. The key SOC design blocks are

1. Microprocessor or microcontroller
2. Counters and timers
3. General purpose IO
4. UART
5. Bus arbitration logic

The memories are discussed in Chaps. 7 and 9 and readers are requested to refer the memory section. The objective of this section is to discuss on the design aspect of these blocks. Finally, these individual blocks can act as an IP and can be integrated together to achieve the desired functional requirement.

The SOC with moderate gate complexity is shown in the Fig. 15.3 and it consists of most of the blocks mentioned above.

### 15.6.1 *Microprocessors or Microcontrollers*

The generalized architecture for processor is shown in Fig. 15.4. As shown in figure it consists of ALU, instruction register and decoder logic, control and timing unit, and program and stack pointer incremental logic. It also consists of bus arbitration logic. While designing the processor it is essential to take care of the design partitioning to code the individual modules using synthesizable constructs. Data path and control path logic need to be partitioned for the better visibility and better

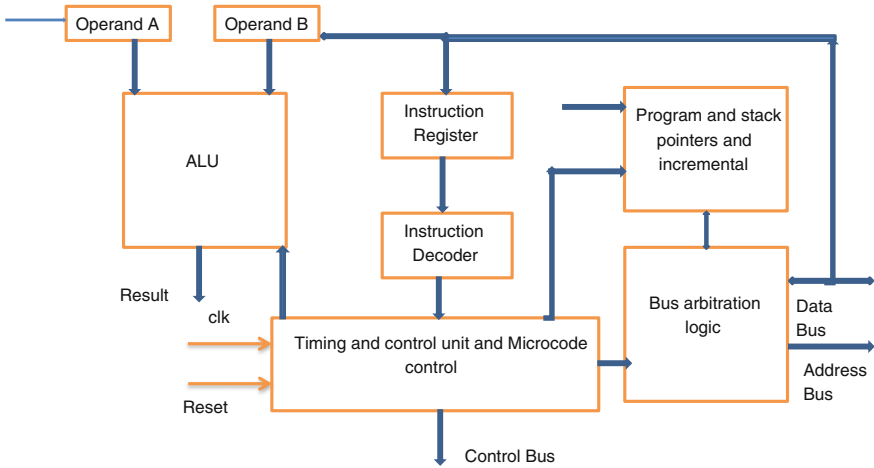


Fig. 15.4 Generalized microprocessor block diagram

timing and performance. Readers are requested to refer Chaps. 1–9 for the RTL coding concepts to design the efficient blocks of processor.

### 15.6.2 Counters and Timers

In most of the design the requirement is to count the predefined number of pulses depending on the external event by using active edge of the clock. An efficient RTL design and functional correctness of the design to achieve the desired performance is the major goal. Consider the block level representation for the timer or counter block shown in Fig. 15.5. The RTL description for the block is shown in the example 15.1.

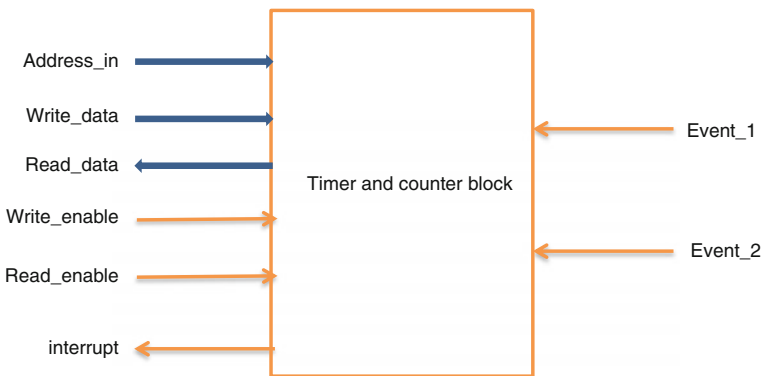


Fig. 15.5 Top level signal diagram for the timer and counter block

### Example 15.1 Verilog RTL for timer-counter block

```
// define the module name
module counter_timer ( clk, Address_in, Write_data, Read_data,
Write_enable, Read_enable, interrupt, Event_1, Event_2 );
input clk;
input [3:0] Address_in;
input [7:0] Write_data;
output reg [7:0] Read_data;
input Write_enable;
input Read_enable;
output interrupt;
input Event_1, Event_2;
reg interrupt_pending, overflow, interrupt_enable;
reg [15:0] reload_counter;
reg [15:0] presale_count, presclae_count1;
reg [15:0] temp_count;
wire operation;
// Write operation functionality

always@(posedge clk)
begin
if (Write_enable && Address_in==0)
interrupt_enable <= Write_data[0];
if (Write_enable && Address_in==4)
prescale_count <= Write_data;
if (Write_enable && Address_in==8)
temp_count <= Write_data;
end
// generation of operation signal high
operation = (Write_enable && Read_enable ==12);

//Read operation
always@ ( * )
begin
if ( Address_in ==0)
Read_data = { interrupt_enable, interrupt_pending};
else if ( Address_in==4)
Read_data = prescale_count;
else if ( Address_in==8)
Read_data = temp_count;
else
Read_data = 0;
end

//Interrupt generation logic
assign interrupt = nterrupt_enable && interrupt_pending;

//timer functionality
always@(posedge clk)
begin
overflow <= (prescale_count ==prescale_count1);
prescale_count <= (overflow) ? (0): (presclae_count+1);
end

always@(posedge clk)
begin
if (overflow)
temp_count<= temp_count-1;
if (temp_count==0)
begin
interrupt_pending<=1'b0;
temp_count<= reload_counter;
end

if (operation)
interrupt_pending<=0;
endmodule
```

Consider Event\_1=1 and Event\_2=0, the write operation functionality initiated by processor is described with 'always' block.

To clear the interrupt write to the Address 12 of the counter\_timer

The read operation functionality initiated by processor is described with 'always' procedural block by using blocking assignment.

Depending on the channel address 'Address\_in' the respective required intermediate value is outputted on output line 'Read\_data'.

Interrupt generation logic is combinational logic and controlled by 'interrupt\_enable' and 'interrupt\_pending' flags.

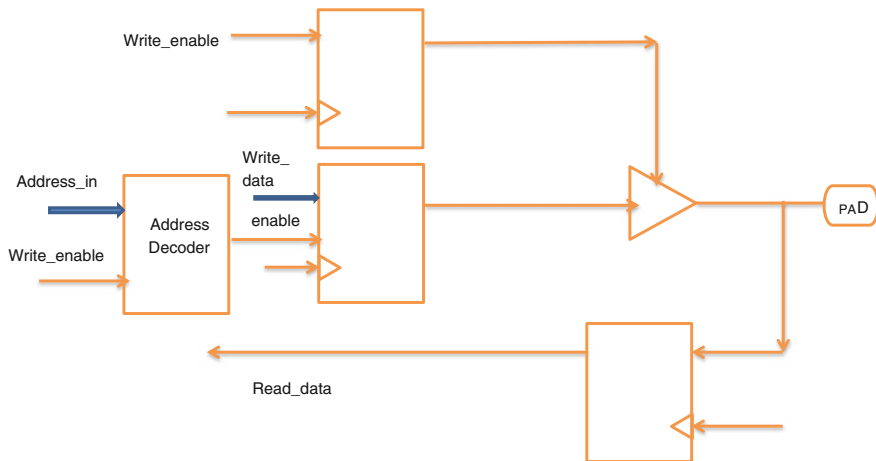
The Timer functionality is described by using the procedural block and sensitive to the active edge of clock.

Depending on the status of 'overflow' flag the temp\_count and interrupt\_pending is assigned.

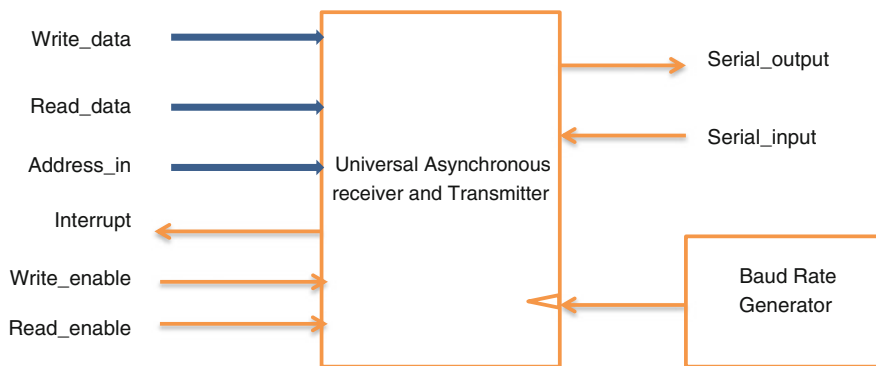
### 15.6.3 General Purpose IO Block

In most of the ASIC SOC design the general purpose bidirectional IOs are used. Multiple IOs are required depending on the required interface inputs and outputs. IOs are used to communicate with the outside world. The generalized structure for bidirectional IO is shown Fig. 15.6.

The partial Verilog RTL is described in the Example 15.2.



**Fig. 15.6** General purpose IO block diagram



**Fig. 15.7** Top level block diagram for UART



```

reg[15:0] ddr_out;
reg[15:0] data_out;

always@(posedge clk)
begin
if (Write_enable && Address_in=0)
ddr_out<=Write_data;
if (Write_enable && Address_in=4)
data_out<=Write_data;
end

//tri-state instantiation

tri_buf U1 ( d_datain[0], data_out[0], ddr_out[0]);
//for the 16 bit IOs there can be 16 more instantiation of the same

tri_buf U16 ( d_datain[15], data_out[15], ddr_out[15]);

//data read

always@9posedge clk)
begin

read_data <= d_datain;

end

```

Double data rate IO structure and the read and write data operations are described in this code.

The Verilog RTL is not complete but can be used as a reference to describe the IO functionality.

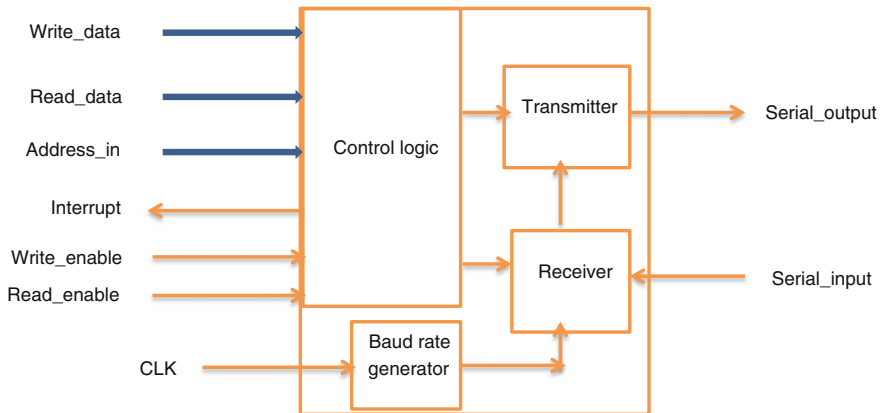
**Example 15.2** Verilog RTL for general purpose IO

### 15.6.4 *Universal Asynchronous Receiver and Transmitter (UART)*

These kinds of blocks can be used in the serial data transfer. The basic protocol is that use the active low start bit and then 8-bit of serial data and finally active high start bit. The data rate can be adjusted by generating the baud clock by using baud rate generator.

The UART consists of transmitter to transmit the serial data using “serial\_output” pin and receiver to receive the serial data using “serial\_input” pin. The data rate is controlled by the baud rate control block. The control logic block can be designed using the multiple data buffers and FIFOs.

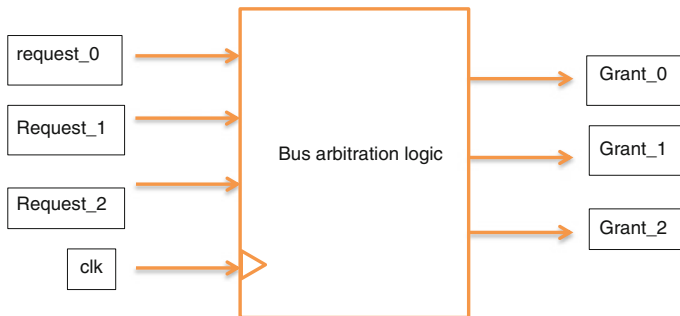
The block level architecture for the UART is shown in Figs. 15.7 and 15.8.



**Fig. 15.8** Architecture diagram for UART

### 15.6.5 Bus Arbitration Logic

The bus arbiters are used to share the same resource by the multiple masters of clients. In the practical scenario typical shared resources are memories, multipliers, and buses. The arbiter decides to which client the service needs to be given, and the property can be static or round robin. The arbiter is shown in Fig. 15.9 and the partial RTL is described in the Example 15.3.



**Fig. 15.9** Top level diagram for bus arbitration logic

```
always@(posedge clk)
begin

    if (reset)
    { Grant_0, Grant_1, Grant_2 } <= 3'b000;
    else
    begin
    Grant_0 <= Request_0;
    Grant_1 <= (Request_1 && (! Resuest_0));
    Grant_2 <= (Request_2 && (! (Resuest_0 || Request_1)));
    end
end
```

The static bus arbitration logic is shown in example and has three requests. The Request\_0 has highest priority and Request\_2 has the lowest priority.

**Example 15.3** Verilog RTL for static arbitration scheme

## 15.7 Summary

The following are important points to summarize this chapter:

1. A SOC is system on chip and consists of multiple processors, IPs, arbitration logic, peripheral interfaces, and protocols.
2. SOC can be prototyped by using FPGA.
3. Third party validated and functional accurate IPs can reduce the overall design cycle time for the SOC design.
4. Hardware and software codesign decides the overall turnaround time for the complex SOCs. The key challenge is the use of the handshake mechanism.
5. Interface timing is one of the critical challenges to achieve in the SOC design.
6. An efficient test plan and verification plan can boost the overall coverage for the SOC design.
7. For complex SOC prototyping multiple FPGA boards can be used to validate the design concept.
8. ASIC migration and porting on ASIC requires the standard cell ASIC libraries with other power aware RTL design tweaking.

# Appendix I

## Synthesizable and Non-Synthesizable Verilog Constructs

The list of synthesizable and non-synthesizable Verilog constructs is tabulated in the following Table

Verilog Constructs	Used for	Synthesizable construct	Non-Synthesizable Construct
module	The code inside the module and the endmodule consists of the declarations and functionality of the design	Yes	No
Instantiation	If the module is synthesizable then the instantiation is also synthesizable	Yes	No
initial	Used in the test benches	No	Yes
always	Procedural block with the reg type assignment on LHS side. The block is sensitive to the events	Yes	No
assign	Continuous assignment with wire data type for modeling the combinational logic	Yes	No
primitives	UDP's are non-synthesizable whereas other Verilog primitives are synthesizable	Yes	No
force and release	These are used in test benches and non-synthesizable	No	Yes
delays	Used in the test benches and synthesis tool ignores the delays	No	Yes
fork and join	Used during simulation	No	Yes
ports	Used to indicate the direction, input, output and inout. The input is used at the top module	Yes	No
parameter	Used to make the design more generic	Yes	No
time	Not supported for the synthesis	No	Yes

(continued)

(continued)

real	Not supported for synthesis	No	Yes
functions and task	Both are synthesizable. Provided that the task does not have the timing constructs	Yes	No
loop	The for loop is synthesizable and used for the multiple iterations.	Yes	No
Verilog Operators	Used for arithmetic, bitwise, unary, logical, relational etc are synthesizable	Yes	No
Blocking and non-blocking assignments	Used to describe the combinational and sequential design functionality respectively	Yes	No
if-else, case, casex, casez	These are used to describe the design functionality depending on the priority and parallel hardware requirements	Yes	No
Compiler directives ('ifdef, 'undef, 'define)	Used during synthesis	Yes	No
Bits and part select	It is synthesizable and used for the bit or part select	Yes	No

## Appendix II

# Xilinx Spartan Devices

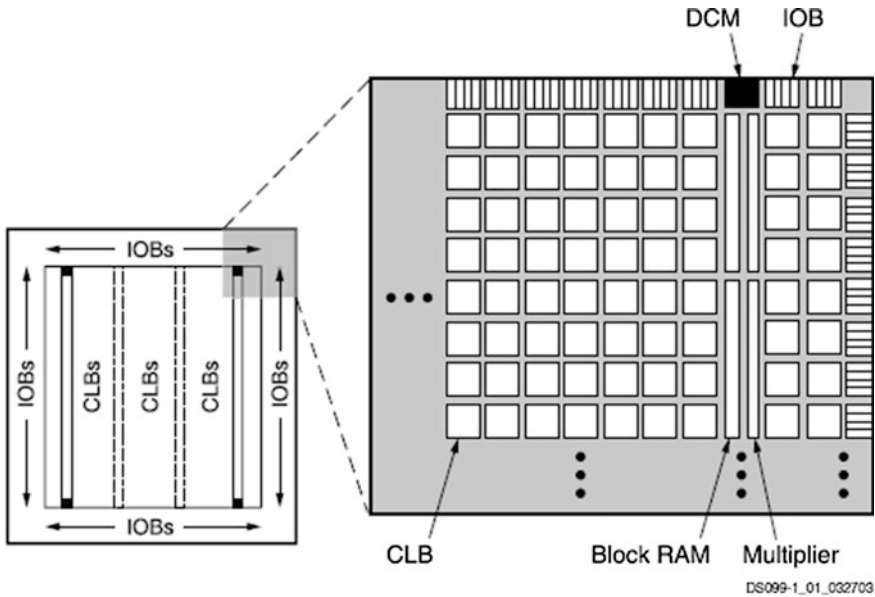
- Xilinx Spartan 3 Devices

Device	System Gates	Equivalent Logic Cells <sup>(1)</sup>	CLB Array (One CLB = Four Slices)			Distributed RAM Bits (K=1024)	Block RAM Bits (K=1024)	Dedicated Multipliers	DCMs	Max. User I/O	Maximum Differential I/O Pairs
			Rows	Columns	Total CLBs						
XC3S50 <sup>(2)</sup>	50K	1,728	16	12	192	12K	72K	4	2	124	56
XC3S200 <sup>(2)</sup>	200K	4,320	24	20	480	30K	216K	12	4	173	76
XC3S400 <sup>(2)</sup>	400K	8,064	32	28	896	56K	288K	16	4	264	116
XC3S1000 <sup>(2)</sup>	1M	17,280	48	40	1,920	120K	432K	24	4	391	175
XC3S1500	1.5M	29,952	64	52	3,328	208K	576K	32	4	487	221
XC3S2000	2M	46,080	80	64	5,120	320K	720K	40	4	565	270
XC3S4000	4M	62,208	96	72	6,912	432K	1,728K	96	4	633	300
XC3S5000	5M	74,880	104	80	8,320	520K	1,872K	104	4	633	300

**Notes:**

1. Logic Cell = 4-input Look-Up Table (LUT) plus a 'D' flip-flop. "Equivalent Logic Cells" equals "Total CLBs" x 8 Logic Cells/CLB x 1.125 effectiveness.
2. These devices are available in Xilinx Automotive versions as described in [DS314: Spartan-3 Automotive XA FPGA Family](#).

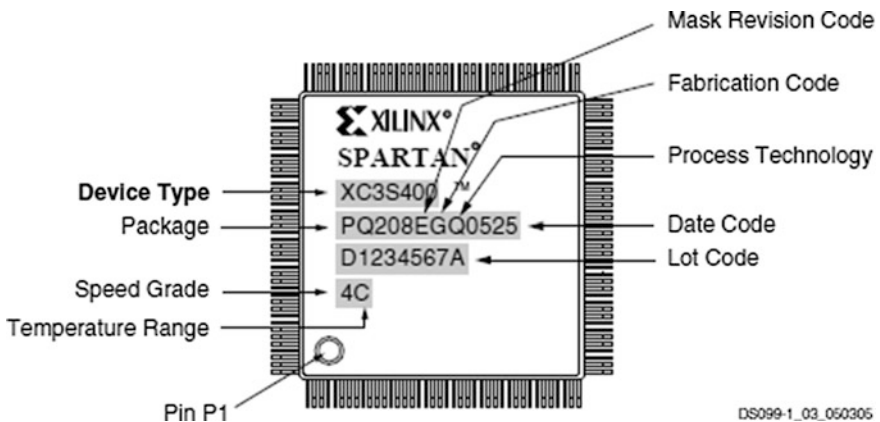
- Spartan 3 Family Architecture

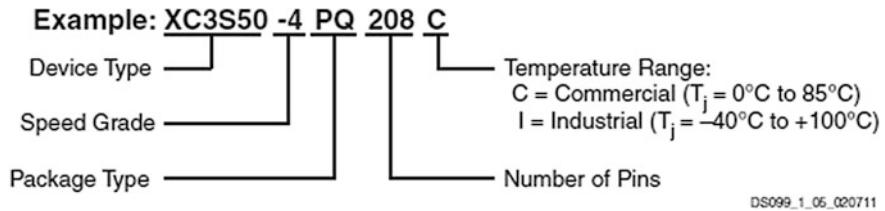


**Notes:**

1. The two additional block RAM columns of the XC3S4000 and XC3S5000 devices are shown with dashed lines. The XC3S50 has only the block RAM column on the far left.

- Xilinx Spartan 3 Package information for Part no XC3S400-4PQ208C





For more information please use the following link [http://www.xilinx.com/support/documentation/data\\_sheets/ds099.pdf](http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf).

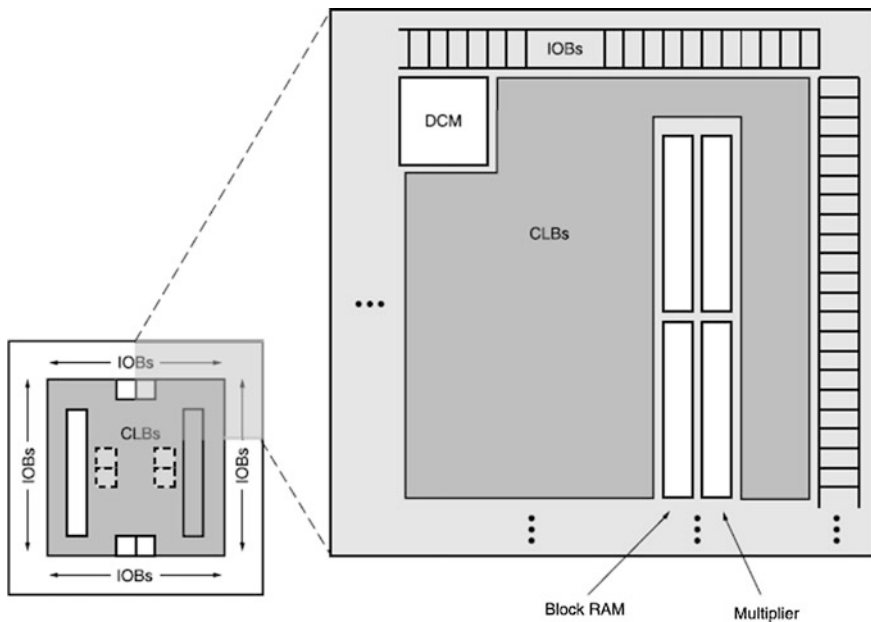
- Xilinx FPGA Spartan 3E Devices

Device	System Gates	Equivalent Logic Cells	CLB Array (One CLB = Four Slices)				Distributed RAM bits <sup>(1)</sup>	Block RAM bits <sup>(1)</sup>	Dedicated Multipliers	DCMs	Maximum User I/O	Maximum Differential I/O Pairs
			Rows	Columns	Total CLBs	Total Slices						
XC3S100E	100K	2,160	22	16	240	960	15K	72K	4	2	108	40
XC3S250E	250K	5,508	34	26	612	2,448	38K	216K	12	4	172	68
XC3S500E	500K	10,476	46	34	1,164	4,656	73K	360K	20	4	232	92
XC3S1200E	1200K	19,512	60	46	2,168	8,672	136K	504K	28	8	304	124
XC3S1600E	1600K	33,192	76	58	3,688	14,752	231K	648K	36	8	376	156

**Notes:**

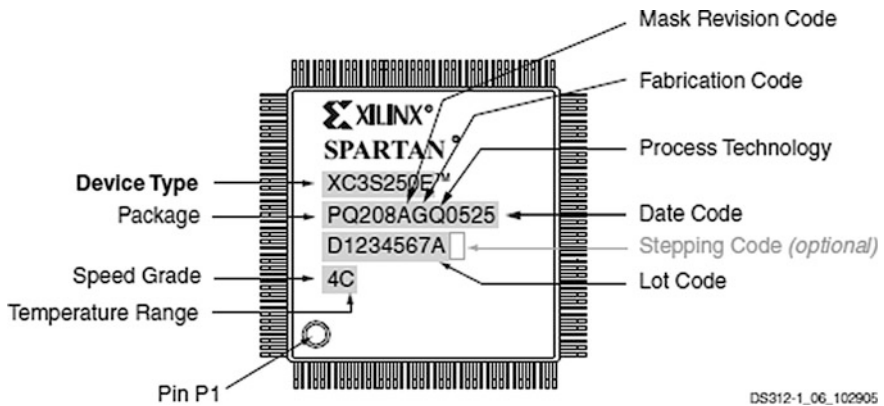
1. By convention, one Kb is equivalent to 1,024 bits.

- Xilinx Spartan 3E Architecture





- Xilinx Spartan 3E package information



DS312-1\_06\_102905



DS312\_03\_062409

For more information please use the following link [http://www.xilinx.com/support/documentation/data\\_sheets/ds312.pdf](http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf).

## Appendix III

# Design For Testability

### **The Design For Testability (DFT) and its necessity is discussed in summarized**

In the practical ASIC design, the DFT is used to find out various kinds of faults in the design. For FPGA designs this step is excluded. The necessity of DFT is for early detection of the faults in the design using scan chain insertions. The functional abstraction of defects is called as fault and the abstraction of the fault is the system level error. Physical testing is carried out after manufacturing of chip to understand the fabrication-related issues or faults.

The defects in the design can be physical or electrical. Physical defects are due to silicon or defective oxide. Electrical defects are short, open, transistor defects and changes in the threshold voltage.

Few of the faults in the design are following

1. Stuck at faults: Stuck at one or Stuck at zero
2. Memory faults or pattern-sensitive faults
3. Bridging faults
4. Cross point faults
5. Delay faults

Testing process is the process of test pattern generation, test pattern application and output evaluation.

Generally, the test flow includes the following:

1. Identify the target faults
2. Test generation
3. Fault Simulation
4. Testability
5. DFT

- Design For Testability (DFT)

The DFT is required to reduce the defect level in the design. Consider the following design; in this design it is not possible to give the test input so design is not testable. The DFT uses the concept of controllability and observability.

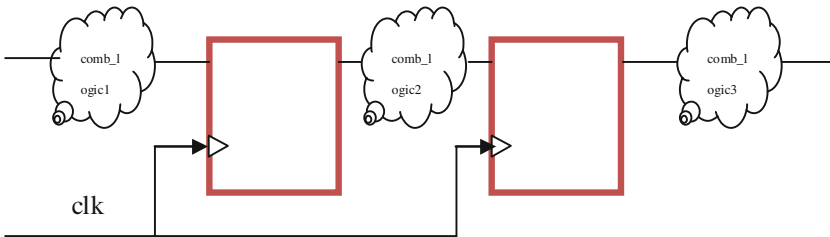
The key steps are

1. RTL design
2. Simulation
3. Synthesis
4. Insert scan chain
5. Layout

If every data input of the register need to be forced to the known value during the test, then the design is controllable.

Observability indicates the ability to observe the node at primary output. The de-sign needs to be controllable and observable.

As shown in the following design, the design input of comb\_logic1 is controllable and the output from comb\_logic3 is observable. But comb\_logic1 and comb\_logic2 are not observable. So for detection of faults, it is essential to make comb\_logic1, comb\_logic2, and comb\_logic3 controllable as well as observable.

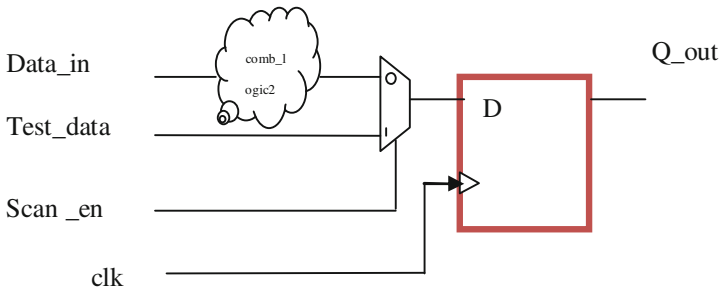


- The basic DFT techniques are: Ad-HOC DFT and Structured DFT. The structured DFT includes the scan-based DFT which is again classified as MUX - based DFT and level-sensitive, element-based DFT. An-other structured DFT technique is MBIST and LBIST. JTAG is used for boundary scan.

Basic MUX-based technique is described below.

- MUX-based scan cell
 

The MUX-based scan cell is shown below and it has additional inputs as Test\_data, Scan\_en. The MUX is inserted at the input of the D flip-flop and during testing Scan\_en=1 the D input is Test\_data. During normal operation, the Scan\_en=0 and Data\_in can pass through the combinational logic to the D input. Thus, the following cell works both in the test and normal modes. The clk can be scan\_clk during the test mode.



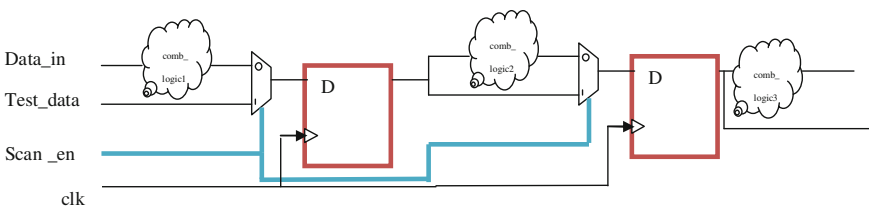
- MUX-based scan chain:

Normally used method is insertion of scan by using MUX logic. MUX-based scan cell shown in the above figure is used to replace the sequential elements from the design. Depending on the requirements the design team decides whether to use partial scan method or full scan method.

In the partial scan method few of the sequential elements are replaced by the MUX based scan cell. In the full scan method, all the sequential elements are re-placed by the MUX-based scan cell.

Due to scan insertion, the area and timing of the design has significant impact. Scan insertion increases the area of the design and due to added MUX-based logic even it affects on the timing of the design. The following example shows the scan chain using MUX-based scan cells.

Most of the time, the partial scan is recommended if area and timing is the constraint but this reduces overall fault coverage. If full scan is used then it increases area and has significant impact on timing but this improves overall fault coverage.



- Scan Design rules

Following are few of the scan design rules need to be considered:

1. Generated clocks in the design: There should not be generated clocks in the design as they are not controllable
2. Combinational feedback loop: There should not be any combinational loop in the design as it creates issues in the timing analysis and hence it is essential to break the combinational loop
3. Gated clocks: Gated clocks need to be avoided as they are not controllable
4. Asynchronous Control signals: There should not be any internally generated asynchronous control signals
5. Do not mix the positive and negative edge triggered flip-flops
6. Avoid use of latches in the design
7. If shift registers are used then do not replace them by using scan enabled flip-flops but only ensure the enable control
8. Do not use the clock input as data
9. Bypass the memories during DFT

# Index

## Symbols

#0delay assignments, 80  
\$display, 229  
\$finish, 229  
\$monitor, 80, 229  
\$strobe, 80  
1 line to 2 or (1:2) decoder, 61  
14 nanometer (nm), 261  
100 K gates, 300  
2 line to 4 or (2:4) decoder, 66  
2:1 MUX, 54  
20 nanometer (nm), 261  
4 line to 16 or (4:16) decoder, 67  
4:1 MUX, 57  
40 nanometer (nm), 261  
50% duty cycle, 266

## A

Acknowledgment or notification, 329  
Active, 80  
Active event queue, 80  
Active power, 364  
Adders, 38  
Addition, 179  
AHB, 386  
Algorithms, 386  
Altera, 242  
ALU, 392  
ALU architecture, 172  
always, 11, 222, 369  
Analyze, 264  
AND, 173  
AND logic, 32  
Antifuse, 234  
Arbiters, 172, 391  
Architecture, 3, 237, 259, 289  
Area, 369  
Area minimization, 30  
Area optimization, 246

Area utilization, 98  
Arithmetic logic unit (ALU), 172  
Arithmetic operations, 11, 39  
ASIC chip designs, 2  
ASIC design, 197  
ASIC library, 389  
ASIC porting, 383  
Asynchronous, 322  
Asynchronous counters, 138  
Asynchronous design, 249  
Asynchronous path, 295  
Asynchronous pulse generator, 169, 248  
Asynchronous reset, 109, 156, 157  
Asynchronous reset 'reset\_n', 200  
Attribute, 308

## B

Barrel shifter, 192  
Basic cell or base cell, 257  
Baud rate control block., 396  
Begin-end, 150  
Behavior, 9  
Behavioral, 7  
Bidirectional, 10  
Bidirectional IO, 395  
Bidirectional shift register, 130  
Binary counter, 114  
Binary encoding, 244  
Binary-to-Gray, 49, 342  
Bit-stream, 238  
Bitwise operations, 16  
Blocking (=), 10, 80, 217  
Blocking assignments, 146, 243  
Block level, 389  
Bluetooth, 382  
Boolean algebra, 2  
Bottom-up, 3, 300  
BRAM, 235  
Buses, 397

**C**

Cadence RTL Compiler, 259  
 Capture flip-flop, 282  
 case, 57, 369  
 case construct, 92  
 case-endcase, 57, 93  
 Case equality, 97  
 Case inequality, 97  
 Cell library, 263  
 Characterize, 312  
 Check\_design, 265, 318  
 Check\_timing, 318  
 Checker, 187, 228  
 Chip level, 389  
 CLB, 232, 235  
 clk initialization, 229  
 clk generator, 229  
 Clock balancing, 370  
 Clock buffer, 370  
 Clock definitions, 286  
 Clock domain, 161  
 Clock domain crossing (CDC), 250, 322  
 Clocked-based logic, 104  
 Clocked logic, 192  
 Clock gating, 251, 363, 364, 366, 390  
 Clock gating structure, 161  
 Clocking boundary, 341  
 Clock path group, 285  
 Clock skew, 242, 387  
 Clock to 'q' delay, 280  
 Clock tree, 363, 364, 387  
 Clock tree synthesis, 260  
 CMOS, 360  
 CMOS logic, 2  
 Code converters, 49  
 Coding guidelines, 79  
 Combinational logic, 10, 27  
 Combinational loop, 245  
 Combinational path, 286  
 Combinational path group, 284  
 Combinational shifters, 192  
 Comparators, 46  
 Compile, 267  
 Compile-characterize, 302  
 Compiler, 314  
 Computational blocks, 386  
 Concentration and replication, 18  
 Concurrent, 10  
 Concurrent execution, 161  
 Conditional assignments, 55  
 Configuration data, 239  
 Consolidated control signal, 334  
 Constant folding, 272  
 Constants, 10

Constraints, 260  
 Continuous assignment, 82, 92  
 Control and timing unit, 392  
 Control path, 165, 392  
 Control signals, 325  
 Coverage goals, 388  
 CPLD, 230  
 CPU, 256  
 create\_clock, 265, 287  
 Cumulative delay, 160  
 Current simulation time, 80  
 Current\_state, 198  
 Cycle accurate, 187  
 Cycles, 390  
 Cycle stealing, 154

**D**

Data arrival time, 278, 281  
 Database, 263  
 Data buffers, 396  
 Data integrity, 322  
 Data path, 280, 364, 392  
 Data path synchronizer, 340  
 Data propagation, 390  
 Data rate, 258  
 Data required time, 281  
 DCM, 235, 242  
 DDR, 241  
 DDR II, 382  
 DDR III, 382  
 Dead zone code, 273  
 Debug, 303  
 Decoder, 58  
 Decrement, 179  
 default, 93, 217, 369  
 Defining hierarchy, 390  
 Delay operators, 185  
 Deserializer, 391  
 Design compiler, 263  
 Design constraints, 5, 300  
 Design environment, 302  
 Design implementation, 252  
 Design object, 264, 301  
 Design partitioning, 274, 306, 346, 386  
 Design performance, 79, 162, 163, 386  
 Design rule constraints, 300  
 Design rule library, 261  
 Design rules, 302  
 Design specification, 257  
 DesignWare, 262  
 Device utilization summary, 239  
 DFT, 259, 389  
 DFT friendly RTL, 390  
 Differential IOs, 387

Different phases, 324  
 DLL, 239, 242  
 DMA controller, 391  
 DRC violations, 315  
 Drivers, 228  
 Drive strength, 302  
 DSP, 192, 382  
 DSP algorithms, 389  
 DSP blocks, 235  
 DSP filtering, 389  
 DUV, 226  
 Dynamic, 278  
 Dynamic power, 251, 360  
 Dynamic voltage and frequency scaling, 365

**E**

EDA, 238, 387  
 EDA tool, 4, 300, 364  
 Edge triggered, 107  
 EDIF, 238  
 Efficient synthesis, 215  
 Effort level, 252  
 Efforts levels, 267  
 Eight-bit parameterized counter, 130  
 Elaborate, 264  
 Empty and full flag, 341  
 Enable, 67, 104  
 Encoder, 68  
 Encoding, 174, 291  
 Encounter from Cadence., 261  
 End point, 284  
 Equality operators, 11  
 Ethernet, 382  
 Even parity, 187  
 Events, 390  
 Exclusive OR, 34

**F**

Fabrication techniques, 258  
 False path, 295, 327  
 Fast debugging, 215  
 Faults, 263  
 Feasibility study, 385  
 FFT, 386  
 FIFO, 386, 396  
 FIFO memory buffer, 338, 345  
 FIR, 386  
 Flash memory, 233  
 Flip-flop, 103, 107  
 Floor planning, 260  
 Four as to one MUX, 56  
 FPD, 230

FPGA, 230, 382  
 FPGA designs, 228  
 FPGA prototyping, 382  
 Frequency synthesis, 242  
 FSM, 197, 244, 314  
 FSM coding, 200  
 FSM control, 340  
 Full adder, 39, 98, 181  
 Full-case, 174  
 Full-case statement, 91  
 Full-custom, 256  
 Full subtractors, 41  
 Functional and timing proven., 386  
 Functional design specifications, 3  
 Functionality, 257  
 Functional model, 3  
 Functional simulation, 238  
 Functions, 172, 184

**G**

Gated clock, 161  
 Gate level netlist, 3, 260, 278, 303  
 GDSII, 360  
 GDSII file, 261  
 General purpose, 387  
 Geometric, 260  
 Glitches, 111, 157, 199  
 Glitches or hazards, 343  
 Glitch free, 216, 244  
 Global clock buffers, 247  
 Glue logic, 274, 306  
 Gray counter, 123, 210, 342  
 Gray encoding, 341  
 Gray-to-binary, 50, 341  
 group, 307  
 group\_path, 310

**H**

Half adder, 38  
 Half subtractor, 41  
 Handshaking, 338  
 Handshaking mechanism, 391  
 Handshaking signals, 329  
 Hazards, 111, 390  
 HDL, 370  
 Hierarchical design, 302  
 Hierarchies, 290  
 High impedance, 11  
 High speed, 257  
 High-speed interfaces, 241  
 High speed IOs, 387  
 Hold, 160



Hold time, 260  
 Hold time violations, 278, 346  
 Hold violation, 280, 315

## I

I2C, 382  
 IC Compiler from Synopsys, 261  
 IEEE 1364-2005, 80  
 IEEE 1801, 371  
 if-else, 57, 369  
 if-else construct, 92  
 IIR, 386  
 Inactive, 80  
 Increment, 179  
 Incremental compilation, 309  
 initial, 11, 220, 222  
 Input and output delay, 267  
 Input argument, 185  
 Input register path group, 284  
 Input string, 185  
 Input to reg path, 284  
 Instantiation, 8  
 Instruction register and decoder logic, 392  
 Inter-assignment delays, 222  
 Interconnect, 232  
 Interface, 258  
 Internally generated clock, 157  
 Intra-assignment delays, 224  
 IO blocks (IOB), 235, 238  
 IO high performance standards, 235  
 IO interfaces, 387  
 IPs, 258, 382  
 Isolation, 371  
 Isolation cell, 365, 371, 390  
 Isolation control, 371

## J

Johnson counter, 127

## L

Latch-based designs, 163  
 Latches, 103, 217  
 Late arrival, 332  
 Late arrival signal, 291  
 Latency, 249, 297, 327, 340, 360  
 Launch flip-flop, 282  
 Layout, 257  
 Leakage current, 360  
 Legal converging, 329  
 Level shifter, 371, 374, 390  
 Level synchronizers, 325  
 Level-to-pulse, 329  
 Level triggered, 154  
 Libraries, 5, 302

Library models, 366  
 Link library, 263  
 Linting tool, 250  
 Load, 302  
 Logical equality, 97  
 Logical flattening, 309  
 Logical inequality, 97  
 Logical operators, 11  
 Logic capacity, 232  
 Logic cells, 238  
 Logic density, 232  
 Logic duplication, 246, 291  
 Logic gates, 27  
 Longer runtime, 305  
 Loss of correlation, 347  
 LUTs, 239

## M

Macrocells, 238  
 Macros, 258  
 Map, 262  
 Map\_effort, 309  
 Master mode, 233  
 Master-slave flip-flops, 368  
 Maximum area, 257  
 Maximum operating frequency, 280  
 Mealy, 198  
 Mealy level to pulse, 200  
 Memory storage element, 107  
 Metastability, 248, 324  
 Micro-architecture, 3, 172, 259, 289, 300, 388  
 Microprocessors, 256  
 Minimization techniques, 27  
 Minimum bus width, 369  
 The min, max corner analysis, 287  
 Min or max, 267  
 Missing 'else' clause, 95  
 Modeling levels, 390  
 Monitor, 80, 228  
 Moore, Gordon, 2, 198  
 Multibit adders and subtractors, 44  
 Multi-bit signals, 338  
 Multi-Cycle Path (MCP), 295, 331, 343  
 Multi phase clock, 165  
 Multiple "always" block, 150  
 Multiple  $V_{th}$ , 364  
 Multiple clock domain, 163  
 Multiple clocks, 274  
 Multiple control signals, 332  
 Multiple Driver Assignment, 102  
 Multiple driver error, 102  
 Multiple drivers, 245  
 Multiple power domain, 382  
 Multiplex decoding, 390

- Multiplex encoding, 292
- Multiplexers, 53, 98, 367
- Multiplier, 235, 386
- MUX, 53
- MUX Synchronizer, 331
  
- N**
- NAND logic, 33
- NBA, 80
- NBA queue, 88, 150
- Negative clock skew, 283
- Negative edge, 104
- Negative level sensitive D latch, 106
- Nested if-else, 244
- Netlist, 238
- Nets, 308
- Next\_state, 198
- Next state logic, 199
- Nonblocking, 10, 217, 243
- Non-blocking (<=), 80
- Non-blocking assignments, 80, 150, 189
- Nonconverging, 328
- Nonsynthesizable, 219
- Non-synthesizable constructs, 11
- NOR logic, 28
  
- O**
- Odd parity, 187
- One-hot encoding, 212, 244
- Opcode, 172
- Operand, 181
- Operand Isolations, 364
- Optimization, 5, 184
- Optimization constraints, 300
- Optimize, 262
- Optimized netlist, 263
- OR, 173
- OR logic, 28
- Oscillatory behavior, 85
- Output register path, 284
- Output to reg path, 284
  
- P**
- Packaging, 258
- Packets, 386
- Parallel and multiplexing logic, 173
- Parallel execution, 386
- Parallel input parallel output logic., 132
- Parallelism, 391
- Parallel logic, 57, 157
- Parameterized binary and gray counter, 123
- Parameters, 10
- Parasitic (RC), 260
- Parasitic capacitance, 361
- Parentheses, 273
- Parity checker, 189
- Parity detectors, 48
- Parity generator, 187
- Partitioning of design, 172
- Path groups, 315
- Performance, 232
- Performance constraints, 257
- Performance improvement, 177
- Phase Shifting, 242
- Photo lithography, 261
- Physical verification, 261
- Pipelined design, 391
- Pipelined processor, 389
- Pipelined register, 163
- Pipelined stage, 313
- Pipelining, 154, 161, 177, 290, 390
- PIPO registers, 132
- PLA, 230
- Place and route, 252, 389
- PLL, 242, 382
- Port interfaces, 306
- Ports, 308
- Positive clock skew, 282
- Positive edge, 104
- Positive level sensitive latch, 104
- Positive slack, 290
- The post-synthesis verification, 246
- Power, 184, 238, 259
- Power compiler, 366
- Power domains, 371
- Power gating, 365
- Power management, 363
- Power planning, 260
- Power rails, 371
- Power Shut-Off (PSO), 365
- Power state tables, 371
- Power switches, 371
- Pre layout STA, 260
- Prime Time, 278
- Priorities, 308
- Priority encoders, 69
- The priority encoding, 292
- Priority logic, 57, 92, 157
- Procedural assignments, 83
- Procedural Block “always”, 146
- Process, 302
- Processing algorithms, 389
- Process node, 360
- Processor, 392
- Program and stack pointer, 392
- Program Language Interface (PLI), 11
- Programmable interconnects, 238
- Programmable logic devices (PLD), 230, 382

Programmable Switch, 232  
 PROM, 230  
 Propagation delay, 280, 313  
 Protocol, 172, 386, 391  
 Prototype, 230, 382  
 Pulse stretcher, 329  
 Pulse synchronizers, 330

## R

Race around conditions, 87  
 Race condition, 85  
 RAM, 140  
 Random test, 388  
 Read, 264  
 Readability, 79, 198  
 Read empty logic, 352  
 Read synchronization, 352  
 Reduction operators, 19  
 References, 308  
 Reg, 83  
 Register balancing, 290, 291, 313, 390  
 Register duplications, 290  
 Register retiming, 390  
 Registered inputs and outputs, 174  
 Register logic, 199  
 Register output logic, 160  
 Register-to-register path, 174  
 Register-to-register timing path, 313  
 Reg to reg path, 284  
 Relational operator, 18  
 Report\_constraints, 315  
 Report\_constraints\_all, 315  
 Report\_timing, 310  
 Reset deassertion, 109, 346  
 Reset recovery, 157  
 Reset synchronizer, 346  
 Reset tree, 156  
 Resource sharing, 47, 98, 174, 246  
 Retention, 371  
 Retention cell, 373, 390  
 Retention control, 373  
 Reusability, 79  
 Ring counter, 125, 226  
 Ripple counter, 140  
 Robust verification, 390  
 ROM, 140  
 Round robin, 393  
 RTL, 3, 259, 306  
 RTL code, 9  
 RTL design, 284, 358  
 RTL synthesis, 3

## S

SATA, 382  
 Scan insertions, 263  
 SDC, 287, 295, 327  
 Sdc commands, 300  
 SDF, 246  
 Search\_path, 263  
 Semiconductor, 383  
 Semi-custom, 256  
 Sensitivity, 10  
 Sensitivity list, 81, 245  
 Sequence detector, 212  
 Sequential, 10  
 Sequential logic, 10, 103  
 Serial data, 396  
 Serializer, 391  
 Set\_clock\_latency, 287  
 Set\_clock\_uncertainty, 287  
 Set\_dont\_touch, 305  
 Set\_input\_delay, 287, 288  
 Set\_output\_delay, 287  
 Set and reset, 370  
 Set-don't\_touch\_network, 369  
 Setup, 160  
 Setup time, 278  
 Setup violation, 310  
 Shift operators, 20  
 Shift register, 127, 130, 146, 240  
 Short-circuit power, 362  
 Sign operands, 13  
 Silicon wafer, 257  
 Simulation, 220  
 Simulation and synthesis mismatch., 90  
 Simulator, 5, 390  
 Single port RAM, 241  
 Skew, 266, 334  
 Slack, 279, 310  
 Slave mode, 233  
 Slowest path in the design, 154  
 SOC, 172, 359, 382  
 SOC components, 391  
 SOC design, 228  
 SOC validation, 387  
 SPI, 382  
 SPLD, 230  
 SRAM, 233  
 SRAM memory cell, 233  
 SRPG, 366  
 STA, 238, 278  
 Standard cell, 256, 360  
 State encoding, 215

- State machines, 274
  - State register, 199
  - State transition, 200
  - State transition table, 210
  - Static, 278, 397
  - Stratified event queue, 80
  - Stratified event queuing, 146
  - Stray capacitance, 361
  - Structural, 7
  - Structural design, 7
  - Structured ASIC, 232
  - Stuck at fault coverage, 389
  - Subroutine, 184
  - Subtraction, 179
  - Subtractors, 41
  - Switches, 371
  - Switching activity, 362
  - Switching power, 161
  - Switch level design, 3
  - Symbol\_library, 263
  - Synchronization, 156, 391
  - Synchronization failure, 324
  - Synchronized asynchronous resets, 250
  - Synchronizers, 274
  - Synchronous, 114, 198
  - Synchronous design, 114, 248, 286
  - Synchronous reset, 111, 157
  - Synopsys, 367
  - Synopsys\_dc.setup, 263
  - Synopsys DC, 300
  - Synopsys Design Compiler, 259
  - Synopsys PT, 278, 300
  - Synthesis, 199, 238, 259, 387
  - Synthesizable, 7
  - Synthesizable constructs, 392
  - Synthesizable RTL, 383
  - Synthesized logic, 200
  - Synthesizer, 259
  - System C, 388
  - System Verilog, 388
- T**
- Target library, 262
  - Task, 172, 184
  - Technology constraints, 258
  - Technology library, 262
  - Temperature, 302
  - Testability, 260
  - Testbench, 5, 219, 390
  - Test plan, 388
  - Test vectors, 388
  - Three-bit down counter, 118
  - Three-bit up counter, 114
  - Three-bit up-down counter, 120
  - Three-procedural block FSM, 200
  - Throughput, 360, 386
  - Time borrowing, 154
  - Time budgeting, 154
  - Time control, 185
  - Time to market, 257
  - Timing analysis, 198, 277, 303
  - Timing analyzer, 286
  - Timing closure, 278
  - Timing goals, 286
  - Timing issues, 247
  - The timing optimization, 308
  - Timing or area, 238
  - Timing parameters, 287
  - Timing paths, 260
  - Timing performance, 238
  - Timing proven IPs, 383
  - Timing sequence, 327
  - Timing summary, 315
  - Timing violation, 160, 174, 260, 280
  - Toggle flip-flop, 200
  - Toggle synchronizer, 330
  - Too tight constraints, 278
  - Top-down, 3, 300
  - Top level, 274
  - Transactions, 390
  - Transistor, 2
  - Transitions in the state diagram, 217
  - Translate, 262
  - Transport and inertial, 390
  - Tri-state, 37, 102, 240
  - Two-level synchronizer, 250
  - Two stage level synchronizer, 109, 157
- U**
- UART, 382, 396
  - Unified Power Format (UPF), 370, 389
  - Unintended latches, 88
  - Unintentional combinational loops, 85
  - Universal logic, 35, 55
  - Unknown, 11
  - USB, 382
  - User constraints, 315
- V**
- Vendor-specific power formats, 370
  - Verification, 259, 343
  - Verification methodologies, 5
  - Verification planning, 388
  - Verilog, 2
  - Verilog IEEE standards, 6
  - Verilog RTL, 3
  - Video decoders, 386
  - Virtex, 239

Virtual clock, [266](#)

Voltage, [302](#)

Voltage level, [374](#)

## W

Weight factor, [310](#)

Wire, [82](#)

Working directory, [263](#)

Write command, [267](#)

Write full logic, [352](#)

Write synchronization, [352](#)

## X

Xilinx, [242](#)

XILINX Spartan, [239](#)

XNOR, [34](#)

XOR, [34](#), [173](#)

XOR logic, [55](#)